Internal Reference Specification for XPL

Begun: 3/98. This page updated 3/13/99. Other pages updated as indicated on the page.
Editing resumes 6/15/2015. Document copied from \XPL\1972-02\XPL.
Main document may be edited or pruned; a New Development chronology will be appended.

Introduction

This document is intended to capture all of the internal details and the development history of the XPL project. It should provide access to the code itself. Everything descriptive in this document will be set in this typeface. Everything that refers to code will be set as follows:

```
0124 LABEL = <expression>
0148       <first indent point>
01A0               <second indent>
01B0                       <third indent>
```

History

XPL was conceived in about 1978, when I built my first desktop computer. It was intended to be a user friendly "high level" assembly language. I wanted to demonstrate that the WYSIWYG concept could be applied to assembly languages, a language genre that was (and is to this day) as cryptic as languages typically get. The key concept in making a programming language WYSIWYG (what you see is what you get) is to make indentation and references part of the syntax. No keywords begin or end a block, indentation itself is the key.

The first version of XPL was written on a Polymorphic computer using a very simple assembler for the 8088. Later, this code was transported to CP/M, and when the IBM PC became available on an 8086 processor, the entire assembler was reimplemented using XPL as the source language. It was then bootstrapped into the PC-DOS environment. From about 1980 to 1995, XPL was upgraded a number of times.

In January of 1996, I began to design a new version of XPL. In July of 2002, I finished it to the point that it supports its own development, and the old version is no longer required. Any future bugs can be fixed using some previous generation of the new XPL to compile the next generation. The new XPL is based on 20+ years of using the old version and thinking about the language in general.

Overview

The new XPL is a family of assemblers—code for any byte-oriented machine can be generated. The default XPL generates code for a .COM file to be executed on an 8086 compatible processor. All XPLs must execute in a COMMAND or DOS window on a PC compatible computer. An object module is limited to 65,536 bytes in size, but this would take about 500 pages of XPL to define. A single XPL source file can't be much over 100 pages long. Twenty pages is where I usually draw the line. In XPL, there is no problem breaking a program into multiple source files. No "header" files are necessary. Symbols may be local to a source file, or global over all source files. Each global symbol is defined once in any module, and it may be referenced from any other module.

XPL stands for Extendable Programming Language. It is a programming language development system. It contains four tools arranged in a hierarchy that is ultimately used to support a 32 bit universal assembly language. These tools have been developed in a 16 bit version of that language. They run as .COM files in a common DOS or COMMAND window. The tools are named: PTRAN, OTRAN, DEXPL, and XPL. Each of these tools is a text to binary translator. An XPL assembler, created from the DEXPL tool, is the most complex of these text to binary translators. In each case, the binary object code is emitted into a .OBJ file according to a standard format. Three additional utilities are used with this translation system. The LINK utility is used to combine .OBJ modules. The PRINT utility is used to print .SRC files. And, the XREF utility is used to create cross reference information in the form of a text file (emitted with a .XRF extension) from any .OBJ file.

Machine Architecture

XPL is appropriate for most modern computer architectures. However, it does not handle addresses or constants larger than 32 bits in any general way. It is also not designed to handle data chunks in sizes other than one, two, or four bytes (each byte being 8 bits). It is designed to handle variable length instructions, and either little or big endian machines.

The architecture of a modern computer (disregarding any that don't fit the above criteria) consists of one or more processors tied together with each other and several levels of memory. Typically, a processor directs the loading of both code and data into itself from memory, and the writing of data back out into memory. Transfer of data between components is typically on a data bus that consists of 1, 8, 16, 32, 64, or 128 parallel wires. There are also address busses that are used to direct the transfer of data.

A "simple" computer has a single processor and one or more memories. Each memory has an address space. One memory (or portion of the address space) typically holds the normal code and data that the program deals with, and another memory (or portion of the address space) is used for input and output. When two, four, or more bytes of data are transferred between memory and an address or data bus, there are two different conventions for doing so. So called "little endian" machines pick up the least significant byte of the multi-byte packet from the lowest memory address. The other convention is the opposite, "big endian" machines pick up the most significant byte from the first memory address. All the XPL tools are implemented in the Intel 8086 instruction set (which is little endian), but they can produce absolute binaries for either big endian or little endian machines.

All computers are designed to go through the following cycle. First, the power is turned on. Second, they are booted. Then, they run continuously until they execute a shutdown procedure. Finally, the power is turned off. If the power is turned off before the shutdown procedure is executed, they may have to perform a recovery procedure as part of their next boot operation. The boot procedure is a program that the hardware automatically executes when the power is turned on. It is contained in a permanent read-only memory.

The boot program prepares an environment for, and loads and executes an operating system. The operating system, in turn, prepares an environment for, and loads and executes various application programs. It also provides an operating interface for the user. The behavior of a computer is a function of the capabilities of its hardware and the programmed software that controls those capabilities.

Software exists in two forms: Source and binary. It is written and accessed by humans in its source form. It is then translated into its binary form so that it can be run on a computer. This translation goes through the following steps: A source to object translation, a linking operation that combines one or more object modules into a form that can be directly used by the computer or its operating system, and finally the loading and execution of an absolute block of binary code. The final step can be accomplished in two different ways: The binary can be written into a read-only memory for direct access by the computer, or it can be loaded and executed by the boot procedure or the operating system. In either case, it must be made present in the computer's general purpose memory, and the processor must be made to branch to its starting point.

All programs are designed to execute in a particular environment of hardware and software. A program's software environment is the subject of the next section. Its hardware environment is a function of the machine's architecture. This consists of the processor's instruction set, the processor's register set, and the address spaces available to the program.

Program Architecture

XPL is designed to make programming in assembly language as abstract and symbolic as possible. The difference between assembly language and higher level languages is that assembly language directs operations in terms of the machine's actual instruction set (operations) and the processor's actual register set (data). Higher level languages have more abstract operations and data that have to be translated into the actual instructions and data for a given machine. A well written assembly language program is often 1/3 the size of a well written program in a high level language (including C or C++), and may run 10 times as fast. This is primarily due to the use of the machine's register set, but it is also due to the simple fact that the processor is utilized more efficiently.

The architecture of a program could refer to one of two things. It could be how the program is organized to do what it does. Or, it could be how the particular form of the program (source, object, absolute) is organized to follow the conventions for a particular machine and operating system. Here, we are going to use the second definition.

Source has the following properties. It is created and accessed by a text editor. It exists as an ASCII text file on a disk. It consists of lines of ASCII text separated by one or both of the ASCII characters, CR and LF, in that order. If the file contains HT characters, they are interpreted as spaces to form columns of eight. If a line ends with any trailing HTs or spaces, these characters will be "trimmed." If a source file contains any characters outside of the hex range 020 – 07E, or other than HT, CR, or LF, these characters will likely cause a syntax error.

Assuming that a source file is not corrupt, it will be rewritten by the XPL translator to indicate the results of the translation. If the file contained no errors, the results of the translation will also include an object file. The object file represents all of the information contained in the source, but in a form that can be linked together with other object files to produce an absolute code image. A source file may be as large as 200K bytes or more. No object or absolute file may be larger than 64K. However, simple utilities can be designed to construct larger binary files from several smaller ones, such that no effective limit restricts the final size or format of an absolute binary file.

All source programs are capable of containing at least the following types of source lines: Comments, meta lines that do not directly result in object code, and instructions that do directly affect or result in object code. Some types of lines may have to appear only before or after other types of lines. Sequence is generally important.

Translator Architecture

XPL is designed around a family of languages and translators that share a common architecture. First of all, they are designed to be executed in a DOS or COMMAND window in an 8086 compatible environment. This dictates the instruction set, the register set, the address space, and the operating system interface that each program expects.

Each program consists of a small boilerplate module that contains the copyright string, a few simple display functions, and a branch to the main entry point. This module also names some of the other modules of the program. All the programs have to load one or more input files, so all reference a utility module called LF (for load file). All have to write one or more result files, so all reference a utility module called SF (for save file). Finally, any that have to load a source file reference another utility module called FF (for fix file).

All source to binary translators in the XPL family branch to a common module called MAIN. This module loads and “fixes” the source file, initializes memory, directs the translation, and does the appropriate wrapup. This module contains two significant functions: MATCH and WRAPUP. MATCH is driven by the data structure that is output from a PTRAN or DEXPL translation, and WRAPUP is driven by the data structure that is output from an OTRAN or DEXPL translation. All the particulars of any translator in the XPL family are contained in modules that are referenced directly or indirectly from MAIN. Direct references from MAIN are to functions in the LF, FF, and SF modules, and to INIT.F (initialize for the entire translation), INIT.P (initialize for the “prep” match), INIT.B (initialize for the “body” match), INIT.T (initialize for “token” translation), NEXT.LINE (get the next line of source), and FINAL (post translation processing). MAIN also references a set of data strings for display and other purposes: FIX.1 is input to the first call to FF (the fix file utility), FIX.2 is input for the post processing call to FF. USAGE, ERRORS, BAD.END, and DONEOK are display strings that report the four alternative outcomes of a translation.

Perhaps the most important references that MAIN (MATCH and WRAPUP) makes to external functions are those that are made indirectly. These references are via the data structures that result from PTRAN and OTRAN translations, and which are linked into the translator via symbols external to MAIN. MAIN references these functions via two address vectors, PRIMITIVES (called by MATCH), and FUNCTIONS (called by WRAPUP). The actual MATCH and WRAPUP data structures are passed to MATCH and WRAPUP via the calls to INIT.P, INIT.B, and INIT.T.

Common Source Format

As mentioned above, PTRAN, OTRAN, DEXPL, and XPL are text to binary translators. These names will be used to reference both the translator program and the source language translated by it. Certain source conventions are common to each translator and will be explained here. First, a standard Text file is assumed. It may be created with EDIT or any editor that can produce an ASCII text file

Each type of source file is given a name with a standard file extension. PTRAN files have a .PT extension; OTRAN files have a .OT extension; DEXPL files have a .DEF extension; and XPL files have a .SRC extension.

All artificial languages have the notions of directive statements, command statements, data statements, and commentary. In the XPL translation system, certain statements are written beginning in column one of a line of text. These include directives to the translator and PRINT, and commentary. These lines are defined by “prep.” In any dialect of XPL, all text that will be translated into object code, plus additional directives to the translator, have the following line format: Five characters are reserved at the beginning of each line. Column six begins the definition field of the line. Column nine begins the body field of the line. These lines are typically defined by the PTRAN “body” definition. Any comment that is allowed to appear on a line containing a statement body must not only follow the body, it must also follow a special syntax defined for that translator. A source file is read, its line endings and tabs are translated into a standard format, each line is scanned and translated, its first five characters are updated, and an appendix, in the case of XPL, is added; then the file is rewritten for review by the programmer.

PTRAN - The Pattern Translator

PTRAN is the initial tool used to build the rest of the system. The source to PTRAN is a metalanguage — a language used to describe languages. It can be used to describe itself and a fairly wide class of other languages. This language is designed to define patterns. Patterns can be translated into a binary data structure and used to drive the MATCH logic of a translator. This is the approach taken with the design of PTRAN itself. The word PTRAN (pee-tran) is used to name both the source language and the translator program.

Here is the definition of PTRAN in the form of its “prep” and “body” text files:

```
0000 prep  '01'= j | name | b
0007 name  = NAME= f e

0000 body           = a d ["'" hex "'"] = alt [sep c alt].. e
0010 alt            = [suc]..
0014 sep            = "|" | "|="
001B suc            = "[" prim [prim].. end | prim
0024 end            = "]" | "].."
002C prim  = "'" hex "'" | g | h | i | a
0038 hex   = hd hd [hd hd]..
003F hd             = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
003F                  8 | 9 | A | B | C | D | E | F
```

Let’s see how these files define the syntax of the very language they are written in. First, notice that each line has three “fields.” The first field consists of five characters (the last of which is usually a space) that are assigned by the translator. The second field begins in column six. It it called the “label” field. The next field is called the “body” field, and it is separated from the label by zero or more spaces. Within the “body” a single space is ignored (unless it appears in quotes), but two or more spaces are the same as a line ending (a comment could follow on that line of text). In these examples of source modules, there are no comments. All of these format conventions are defined by the language of the example itself.

Notice the first label of each source file. The “prep” module begins with “prep” and the “body” module, with “body.” PTRAN is coded to create a single entry point equal to the label on the first “body” line of the source file. The label on the first “body” line may not be referenced by any other pattern. The label on each subsequent pattern must be referenced by some pattern definition above it. Otherwise, a label error is written into the source.

Let’s try to “read” the first line of “body.” Skipping to column 6, it reads roughly as follows: The pattern “body” is defined as a call to the “a” primitive, followed by a call to the “d” primitive, followed by a bracket sequence, then by two more pattern elements, another bracket sequence with two dots suffixed to it, and finally by the “e” primitive. Each of the brackets encloses three pattern elements.

The purpose of a pattern definition is to MATCH some source text and drive the translation of it into a sequence of tokens. Afterward, the tokens are translated by a WRAPUP pattern into a sequence of binary object code. To see how this is done, you have to know what all of the pattern elements are, and what they can do. A single alpha character represents a call to a hand-coded “primitive” that performs some function on behalf of the match. For example, it may recognize a particular type of symbol and produce a token when successful, or fail and cause MATCH to try another alternative. In the “body” pattern, the following single characters appear: a, c, d, and e. These primitives (which must each follow certain coding conventions to be compatible with MATCH) are defined as follows:

“a” matches a lower case alpha followed by one or more dots, digits, or lower case alpha, with the proviso that it stops short of any trailing dot (the dot isn’t necessarily an error, but it isn’t matched by “a”).

“c” handles continuation, if present, and always succeeds.

“d” matches zero or more spaces, and always succeeds.

“e” matches the end of a line defined as two or more spaces or an end-of-line character.

Forgetting about the bracket sequences, the “body” pattern has three other types of pattern elements. It has references to subpatterns, indicated with two or more alpha characters (exactly what “a” is designed to match), it has a quoted literal (" ' "), and it has an unquoted literal (=). The rest of the syntax involves the brackets. The first brackets enclose a sequence that is optional. That is, it may or may not appear in the text. If it does, it is matched and a corresponding token sequence is produced. If it is missing, that’s OK, too. A token is produced to indicate that the sequence was not matched, and the match continues at that point. The second bracket has two dots suffixed to it. This indicates “zero or more.” The enclosed sequence is not only optional, but it can be repeated as many times as the author of the source might require.

Now, for a mental exercise: Imagine matching the first line of “prep” with the pattern defined by the first line of “body.” Starting in column 6, of “prep” and matching the first pattern element “a” of “body” we find that the “a” primitive is coded to match the string “prep.” Next, “d” matches the whitespace between “prep” and the quote character. Next in the pattern is the bracket sequence [" ' " hex " ' "]. This matches the next four characters of “prep” ('01'). The quotes are matched literally, and “hex” invokes two levels of subpatterns that match the characters (01). Next, the literal = matches the = in the source. Now it gets more complicated. The next pattern element is the subpattern “alt” and the next source is a reference to the “j” primitive function. An “alt” is defined as zero or more “suc” patterns. And, each “suc” is defined as something beginning with a literal “[“ character, or a “prim” subpattern. The vertical bar (|) is read as “or.” Notice that “prim” is defined as hex in quotes, or alternatively, one of the primitive functions (in the order that they are attempted): g | h | i | a. The definition of “a” was given above. The others are as follows:

“g” matches "non-double-quote" or 'string"with"double-quote' (quotes are used to enclose metacharacters).

“h” matches all unquoted literals, stopping short of any of the following: " ' [|] CR, space, or lower case alpha.

“i” matches a primitive (a single lower case alpha, not followed by any lower case alpha, dot, or digit).

Bottom line: the “j” primitive in the source is matched by the “i” alternative of the “prim” subpattern. This is all that can be matched by that alternative. The next source to be matched is | and this causes MATCH to pop back three levels to where “alt” was referenced in the “body” pattern. The next element (inside repeat brackets), is the “sep” subpattern. “sep” is defined as the literal | (possibly with an = after it), so it matches. The next source is “name” which matches the “a” alternative of the “prim” subpattern. But, before that the primitive function “c” is called. This handles continuation from one line of source to another. In this case, there is no continuation, so “c” merely succeeds without doing anything. Now, “alt” references “suc” which references “prim” which has “a” as one of its alternatives, and “a” matches “name.” The final two symbols of “prep” (| b) are matched by a second application of the repeat bracket, and the end of the source line is reached. The line ending is matched by the “e” primitive, and the entire operation is returned from MATCH as a success.

Each line of a source file is first matched by the “prep” definition, then by the “body” definition (if the “prep” match fails). There are no hard and fast rules as to how the syntax and functionality of these two differ, but, in general “body” statements begin in column 6 and “prep” statements begin in column 1. “prep” statements often have to appear in a certain sequence, perhaps no more than once in a source file, so there may be logic coupled to which “prep” alternatives have matched during a translation. In the “prep” definition, above, the following primitive functions are referenced.

“b” succeeds if the line appears to be a “prep” statement; it fails if the line appears to be a “body” statement.

“f” matches “file” characters — characters valid in a filename.

“j” matches any line to be ignored, such as a comment line, or printer directive.

The “j” primitive is the first call made when any new statement is matched. It handles some aspects of initialization, and succeeds when any comment or printer directive is detected. When it succeeds, no further matching of the line can take place, nor is the line changed in any way.

The “name” pattern is used to define the filename of an object module. Normally, it’s the same as the name of the source file and acts as little more than a comment. Without a NAME statement, the object module is given the same name as the source module. However, if differently named source files are to have object files with the same name, the NAME statement is the way to make this occur.

The only way for the primitive “b” to be called is for both “j” and “name” to fail. In this case, the line may be a normal line of source, and needs to be matched with the PTRAN “body” definition. However, the line may simply be a failed “prep” statement. If this is the case, “b” needs to succeed. Otherwise, “b” initializes for the matching of a normal statement, and forces this to occur by failing. This business of a match succeeding or failing brings up the logic of pattern matching in general, and this needs to be discussed next on the way to a full understanding of PTRAN. First, we will look at the PTRAN pattern data structure, then the matching algorithm driven by it.

The PTRAN Pattern Data Structure

The PTRAN “match” function is handed a specific pattern data structure by the “init” routine that is called immediately before “match” is called. This structure can be linked into any of the translators (PTRAN, OTRAN, DEXPL, or any version of XPL), because they all use the same “match” function. The form of the pattern data structure (defined as XPL “code”) is as follows:

```
<name>         ENTRY
               =@main
               =@sub1
               =@sub2
               . . .
main: =<byte string>
sub?: =<byte string>
               . . .
```

The <name> of this structure is given in the first pattern definition (“prep” would define PREP). The structure begins with an index to the “main” pattern definition, then a vector of indices to each of the subpattern definitions.

The subpattern vector is followed by the byte string that defines the main pattern, and that is followed by the subpattern definitions themselves. Here is how the <byte string> definitions are laid out:

```
                 head [override] [suc].. [next [suc]..]..
```

Bytes in the "head" and "next" positions are interpreted (in binary) as follows:

```
      head = 1xssssss, where ssssss = skip to next, and
                 x = 0 means byte following is a suc byte
                 x = 1 means byte following is an override
      next = 0xssssss, where ssssss = skip to next, and
                 x = 0 means this alt # is one more than the previous alt #
                 x = 1 means this alt # is the same as the previous alt #
```

Each "suc" (or successor pattern element) is coded (in hex) as follows:

```
      00-1F      is an override code whose interpretation depends on context.
                 Following the head byte, it's the initial alt # of a
      bracket.
                 As the last element of an alt, it defines the # of that alt.
                 Elsewhere, if = current token pointer, then save the pointer
                            else, set current token pointer to the override.
      20         Open Bracket (Option or Repeat); if Open, then Close Option.
      21-60      Match a literal byte = suc code.
      61-7A      Call a primitive function (codes a - z)
      7B-7E      Match a literal byte = suc code.
      7F         Close a Repeat Bracket (if not Open, then NOP)
      80-FF      Match a subpattern, using suc code as index to pattern
      vector
```

Notice that all printing ASCII characters represent themselves as literal bytes to be matched, except lower case alpha, which represents primitive functions to be called. "match" never distinguishes case when matching an alpha literal, so lower case source matches an upper case literal. The codes 80-FF represent embedded subpatterns. The code 80 corresponds to @Sub1, 81 to @Sub2, and so on. This is pretty straightforward, so far. The Repeat Bracket is also fairly easy. Although repeats cannot be embedded within repeats, this can be done by using a subpattern reference and defining the nested repeat as the subpattern.

The complex stuff involves override codes. As indicated above, override codes are subject to four different interpretations. Remember the optional pattern that appears to the left of the "=" in a "body" definition? This permits us to define the override code that may follow the "head" byte in the pattern data structure. Now, what does this override do? First, it turns the whole definition into a "bracket." Second, it defines the starting alt # for the pattern. In this context, the override may be any value 00 - FF. When an override appears as the last "suc" in an "alt," it sets the alt #. What difference does an alt # make? In XPL it is the most common way to define the value that will be plugged into a field within an instruction. So, it is important to define patterns ordered in such a way as to give the alt # the correct value, and when order must be changed, explicit control over the alt # is necessary.

When an override appears elsewhere, there are two cases: The default is to set the current token number. If the override is equal to the current token number, then a pointer is saved to the current token. This enables access to the string about to be matched by the current pattern alternative. It can be used to match a copy of this string later in the pattern. These form the basis for two important features of XPL (and could be used similarly in other languages). The first is the XPL construct exemplified by "reg=reg+1" meaning that "reg" is to be incremented. If "reg" were any of the symbols A, B, C, D, or E, we would want to allow only A=A+1, B=B+1, and so forth. A=B+1 is not to be allowed. The PTRAN for this would be something like:

```
      0000 bump  = '01' reg = a + 1
      0007 reg          = A | B | C | D | E
      << a - is a primitive that does a rematch of the saved token pointer
      >>
```

The second feature is the ability to set the token output pointer. By default, tokens are emitted one after another in a variable length sequence. Often, many of the tokens aren't interesting, and it may be desirable to put the ones that are interesting in fixed positions (for an easier "wrapup" definition).

There are only 26 primitives allowed. If you need more, you may follow a primitive with an explicit byte code (coded & looking like an "override code"), and let the primitive decode it and branch to several sub-primitives. There's an example of this on page 15. Sometimes "override" codes must be followed by 7F (no operation) codes. This is because, otherwise, they would be the final codes in the alternative and would be interpreted as Alt Overrides. Elsewhere in a pattern, the primitive interpreting the code could prevent "match" from seeing it, but in the final position an override code is subject to being deleted in the translation process (Alt Override codes are automatically inserted where alternatives are reordered and removed where they are redundant).

The PTRAN Pattern Matching Algorithm

The heart of PTRAN is contained in the MAIN source module of PTRAN. MAIN is used, unchanged, by the other programs: OTRAN, DEXPL, and XPL. This module contains calls to initialize a translation, calls to finalize a translation, and the loop of code, below, that drives the translation. This loop begins with a call to initialize the "prep" phase of a match. That is, the part that recognizes the meta-text. Since this phase is preparatory to matching the main syntax, it is called "prep." The first call to "match" is with a handle to the "prep" data structure. After this initial call to "match" all subsequent calls come after a call to "init.b" to initialize the "body" phase of the match. Within the entire loop, the CY flag is used to signal failure and NC to signal success. The "error" routine is called when "init.b" has determined there are no more pattern data structures to be tried. In the case of PTRAN itself, this occurs after the "body" syntax has been tried. In the case of more sophisticated translators, there might be several alternative "bodies."

When any "match" succeeds, the "init.t" and "wrapup" functions are called. Whether the inner loop exits via "error" or "wrapup," the "next.line" function is called to determine whether the translation is complete, or whether the loop is to be continued with another call to "init.p" followed by the first call to "match" the next statement. Notice the loop structure. The inner loop is continued via the conditional statement that tests the result of "init.b." The outer loop is continued via the loop statement that test the result of "next.line."

```
DO      CALL INIT.P                         unique function
        DO      CALL MATCH                  common function
                IF CY,
                        CALL INIT.B         unique function
        *               IF NC,AGAIN
                        CALL ERROR          unique function
        *               NEXT
                CALL INIT.T                 unique function
                CALL WRAPUP                 common function
        CALL NEXT.LINE                      unique function
LOOP UNTIL CY
```

The above code is a sample of the XPL assembly language used to code the entire XPL system, and which is translated by the resulting XPL assembler. There are four branch statements in this code segment, yet no statement bears a label. Since this document is for advanced programmers trying to learn the inner details of the XPL system, a complete knowledge of XPL will be assumed from here on.

However, a few words describing register conventions are in order at this point. When a .COM file is executed, all of its segment registers are equal to the CS register. SP works backwards from the top of this segment. Each program in the XPL system follows this convention. The code segment and stack segments are identical; SS is never modified. When SP gets sufficiently close to the contents of E.O.MEM(2), an index to the next available byte of freespace, the program aborts due to a memory overflow. The segment above CS is often referenced by DS, and contains the source file. Above that is the cross reference segment, the symbol table, the token segment, and at the top of memory, the object segment. It should be mentioned that the source segment may be more than 64K and DS may be advanced through it. No MATCH or backup may exceed the 64K limit, however. The symbol table, object, token, and cross reference segments are also limited to 64K each.

It is instructive to see how the "init.p" and "init.b" functions set up the registers for "match." The key registers are DE which must index (in the CS space) the base of the subpattern vector (@Sub1 in the pattern data structure), BP which indexes the beginning of a pattern definition (also in the CS space), SI which indexes the next byte of source in the DS space, and DI which indexes the next token to be emitted into the ES space. BC is initialized by Match itself (and must be preserved by any primitive match function). AA and HL are used as scratch registers by Match and any of its primitives.

The object of "match" is to emit tokens into the token segment. These tokens fully characterize a complete source statement (translating most LALR(n) languages into simple LL(1)). The "prep" and "body" data structures represent the allowable syntax, and each drives an attempt at a match. When "match" succeeds, a "wrapup" is invoked. This is actually another syntax driven (LL(1)) translation. This time, instead of the source being defined, the binary object is defined by the OTRAN metalanguage. Notice the circularity involved: The front end of PTRAN is defined by its "prep" and "body" definitions in PTRAN. Given a pattern data structure, the token output is also defined. The back-end of PTRAN is defined in OTRAN. This is to complete the translation of the tokens into the final object code, or .OBJ file. OTRAN is defined in the same way. PTRAN definitions define its front end, and OTRAN definitions define its back-end. Likewise for DEXPL.

OTRAN - The Object Translator

OTRAN is not an elegant language, nor is it easy to read. It's even harder to write it! Having said that, it's better than writing code in any general purpose language to accomplish the same objective. First, we'll look at the metalanguages. Then we'll define the objective of OTRAN. Last, we'll see how its objective is accomplished.

The PTRAN source that defines the OTRAN metalanguage is as follows:

```
0000 body  = [a:] d list [end] e
000B list  = cond [, cond].. d [prim].. | [prim]..
0019 end   = . | ; | :a
0020 cond  = $< hh | $= hh | $> hh | $ hh = hh [,hh]
0035 prim  = EMIT() | EMIT(hh [,hh] [,hh] [,hh] [,hh] [,hh]) |
0035            XOR(hh) | XOR($ [hh] [,hh] ) | i | a | < | - | > | +
0079 hh    = hd | hd hd
0080 hd              = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
0080                   8 | 9 | A | B | C | D | E | F
<< primitives
          i = any single lower case alpha or ` { | } ~ ? characters.
   The a, b, d, e, f primitives and the PREP definition are the same
   as in PTRAN.  Primitives c, g, & h are not referenced in OTRAN.
>>
```

The OTRAN source that defines the back-end, or "wrapup" phase of (the above) OTRAN is as follows:

```
0000 back: $=0     def list end;
0008               $=1     ;
000A               $=2     + f;
000E                       e
000F def:          d+.
0012 list: $=0     cond rep.c rep.p;
001A               $=1     rep.p.
001E end:  $=0     EMIT();
0021               $=1     EMIT(08);
0025                     + EMIT(0E) a+.
002B cond: $=0     EMIT(30) b;
0030               $=1     EMIT(40) b;
0035               $=2     EMIT(50) b;
003A               $=3     EMIT(10) b.
003F rep.c:                cond :rep.c
0043 rep.p:                prim :rep.p
0047 prim: $=0     EMIT(07);
004B               $=1     EMIT(01) b;
0050               $=2     EMIT(0D) b;
0055               $=3     EMIT(80) b;
005A               $=4     c+;
005E               $=5     EMIT(F) a+;
0064               $=6     EMIT(9);
0068               $=7     EMIT(A);
006C               $=8     EMIT(B);
0070               $=9     EMIT(C).
```

The OTRAN source that defines the back-end, or “wrapup” phase of PTRAN, is as follows:

```
        0000 back: $=0     body;
        0004               $=1     ;
        0006               $=2     + f;
        000A                       e
        000B body:         d + < EMIT(80) mark + alt > c rep.alt `
        001A mark:         XOR(40) hex .
        001F alt:          suc :alt
        0023 suc:  $=0     opt.prim;
        0027               $=1     prim .
        002B opt.prim:             EMIT(20) prim rep.prim end .
        0034 rep.prim:             prim :rep.prim
        0038 prim: $=0     hex ;
        003C               $=1     g + ;
        0040               $=2     h + ;
        0044               $=3     i + ;
        0048               $=4     a + .
        004C end:  $=0     EMIT(20) ;
        0050               $=1     EMIT(7F) .
        0054 rep.alt:              < EMIT() sep + alt > c :rep.alt
        005F sep:  $=1     XOR(40) .
        0063 hex:  + b ++ bb.
        006A bb:           b ++ :bb
```

Thus, we have now defined the source and object of both PTRAN and OTRAN in the PTRAN and OTRAN metalanguages. Now, we need to examine the definition of the OTRAN metalanguage to better understand the token to object translation it defines. Once both PTRAN and OTRAN are fully understood, we can move on to DEXPL, which defines a set of “body” data structures that drive “match” and the “back” data structures that drive “wrapup” within an XPL assembler. DEXPL is a single metalanguage (and translator) that does both of these tasks. Its PTRAN definition runs over 30 lines of source, and its OTRAN definition runs over 80 lines. Finally, we will look at the DEXPL source that defines XPL. The source for an XPL definition can run a dozen pages or more.

The OTRAN Metalanguage

The main pattern definition for OTRAN (as given on the previous page) is as follows:

```
        0000 body  = [a:] d list [end] e
```

This indicates an optional label followed by a literal colon (the “a” primitive followed by a colon appears inside a pair of brackets). Next on a line of OTRAN may be some whitespace (indicated by the “d” primitive). Then the pattern element “list” occurs, followed by an optional “end” pattern, and a mandatory end of line (the “e” primitive).

```
        000B list  = cond [, cond].. d [prim].. | [prim]..
```

“list” is the body of an OTRAN statement. It consists of one or more “cond” patterns followed by optional whitespace, followed by zero or more “prim” patterns. Or, alternatively, no “cond” patterns followed by zero or more “prim” patterns (in otherwords “list,” and therefore the whole body, may be empty).

```
        0020 cond  = $< hh | $= hh | $> hh | $ hh = hh [,hh]
```

The “cond” pattern always consists of a “$” followed by a less than, equal, or greater than sign, or by a hex value “=” and one or two more hex values. The various ranges allowed for the hex values represented by “hh” are not the same everywhere. “hh” may never be greater than 0FF, but in some cases it must be less than 0F. These limits will be discussed later.

```
        0035 prim   = EMIT() | EMIT( hh [,hh] [,hh] [,hh] [,hh] [,hh] ) |
        0035              XOR(hh) | XOR($ [hh] [,hh] ) | i | a | < | - | > | +
```

There are ten alternatives defined in the “prim” pattern. Basically, they allow from one to six specific byte constants to be emitted into the object code, they allow simple modification of the most recent byte emitted via the use of an XOR operation, they allow a primitive function to be referenced (with the “i” primitive), they allow another OTRAN definition to be embedded into this one (with the “a” primitive), and they allow four other simple operations to be specified (with the literals < – > +). Notice, above, the definition of the “i” primitive in OTRAN includes not only the 26 lower case alpha characters, but also ` { | } ~ ? for a total of 32 primitive functions. If you need more, you may follow a primitive with EMIT(codes). An example of this appears on page 34.

The Objective of OTRAN

OTRAN is a language and translator for describing and producing the binary output, or object code, for any translator in general. PTRAN defines a translation of source to intermediate tokens, and OTRAN completes the process by defining the token to object translation. Thus, the input to OTRAN is the output of PTRAN, and the OTRAN must be coupled perfectly with the PTRAN tokens, or bugs in the translator will be the result. Let's proceed from the simpler to the more complex.

PTRAN Intermediate Code - Tokens

When PTRAN patterns are matched with source, the progress of the match is represented by tokens. If the match fails at any point, both the source and token pointers are backed up and an alternative, if one exists, is tried. Only when no alternatives remain does a match fail. Assuming that some legitimate interpretation of the source exists, then a sequence of tokens will represent it.

Each token consists of four bytes: The first byte is the number of the alternative that matched at that point. The second byte is the length of the string matched by that alternative. The next two bytes are the (word) index into the source of the beginning of the string that matched. If the alternative indicated was not a primitive, then the token delimits a string matching more than one symbol. Generally, only the number of the alternative (alt #) is referenced for such a compound token, the length and source index are ignored (which is good, because the length might be 255, representing all strings of 192+).

The alt # in a token rarely gets very large. An initial byte of 255 is used to flag a special token. This form of token is used to represent the matching of a Bracket construct. That is, a pattern defined with an override code to the left of the "=" character, or any construct within a definition where elements are enclosed in brackets. These constructs are all represented by the code 255 in the initial byte of a token. The next byte, instead of being the byte count of the source string matched, is the override code (for a pattern bracket) or 255 (for an option or repeat bracket). The (word) index that follows is the index of the next token following any produced by matching the bracket construct. In other words, it indexes the token segment, not the source segment. It delimits the set of tokens emitted by matching the elements of that bracket. This enables both option and repeat brackets to be handled in the same way. And, it gives a handy way of recording the end of the tokens matching this variable construct.

Producing the PTRAN Object Code

We have seen the definition of PTRAN in itself and we have studied its pattern data structure. The PTRAN for PTRAN, and the OTRAN for PTRAN, are coupled into a translator that produces this object code. Now, let's take a closer look at the OTRAN definition that defines the second stage of this translation. OTRAN defines a pattern match much as PTRAN does. In this case, however, it is the intermediate sequence of tokens that is being matched. There is no provision (nor any need) for an OTRAN match to fail or back up and try another alternative. Each decision is made when it is first encountered. This is easy, because "match" has translated a wide class of languages and expressed them in simple LL(1) form.

OTRAN patterns consist of conditional and unconditional pattern elements. Conditionals match the alt # of the current (or an absolute) token. If successful, the pattern elements following the conditional are executed until a ; or . is encountered, at which point "wrapup" returns to a higher level or back to the main translation loop. If a conditional does not match, "wrapup" skips elements until it encounters a ; or . element. A ; causes it to resume execution of pattern elements. A . causes it to return to a higher level (or exit "wrapup" altogether).

Notice (below) that the first four lines of OTRAN form a single OTRAN pattern. Technically, the final line should end with a . character. The reason it doesn't is that the "e" primitive is used to report an error and force a return directly to the main loop. Errors detected in "wrapup" are generally called semantic errors.

```
0000 back: $=0     body;
0004               $=1      ;
0006               $=2      + f;
000A                        e
```

This pattern begins with the conditional $=0 which means if the alt # of the next token is equal to zero, it should invoke "wrapup" with "body" as the pattern. Otherwise, if $=1 it doesn't do anything (the ; indicates the end of the conditional clause). If the token equals two, it should perform the + operation, then call the "f" primitive. Finally, if none of the above, it calls the "e" primitive to return an error.

At this point it might be useful to explain what some of the built-in operations are: The + operation causes "wrapup" to advance the token pointer by one token. The – operation causes it to move the current EMIT pointer back by one byte. The < operation causes "wrapup" to stack a copy of the current EMIT pointer. The > operation pops this pointer for use by the next primitive function (this pop is into certain CPU registers, it does not change the current EMIT pointer).

If you were wondering why the OTRAN pattern given above had conditionals for alts numbered 0, 1, 2, and "other," it is because only one OTRAN pattern data structure is defined. It needs to join the results of whichever front end structure was matched. Thus, if "body" is matched, the initial alt # will be zero (by default). If "prep" is matched, the alt # of the first token will be one, two, or three, since a "01" override is used to begin that definition. If no override is used to define the initial token number in a PTRAN definition, a standard token is emitted instead of a bracket token (and alts begin with number zero).

Now assume that "wrapup" finds $=0 in the "back" pattern (see previous page) and invokes "body" (below). Let's follow the logic from there. In "wrapup" as well as in the OTRAN definitions, a whole different set of primitives exists. We need to know how they are defined before we can very well understand the OTRAN. The "e" and "f" primitives were mentioned in the definition above. Here is the full set:

```
<< the OTRAN primitives for PTRAN
           ` - fixup an entire definition (post processing)
           a - emit a <label> reference
           b - handle & emit hex byte(s) into the object file
           c - fixup the previous alt prefix
           d - define a <label>
           e - report an error and return directly to main
           f - put NAME=<filename> into the symbol table
           g - emit a quoted literal
           h - emit an unquoted literal
           i - emit a function call (a..z)
>>
000B body: d + < EMIT(80) mark + alt > c rep.alt `
```

The first token of a "body" is the label being defined. This token is handled by calling the "d" primitive. The + operation moves the token pointer forward one token. The < operation pushes a pointer to the object code where EMIT(80) is about to occur. Next, the hex value 80 is emitted into the object code. This byte is the first byte of the pattern about to be produced. The next token to be considered is the optional override byte. This is matched by the OTRAN reference to "mark." Now, a subtle thing must be explained. It has to do with the token pointer. When a bracket token is matched by a subpattern reference, as it is here, there are two possibilities: First, the bracket may be empty. It might not have matched anything. Only if the bracket is not empty will a recursive call be made to "wrapup" the subpattern. If the bracket is empty, or if it is an option or repeat bracket, the token pointer is automatically advanced. Bracket tokens were specifically designed to be matched by an OTRAN subpattern reference. In the case of a non-empty pattern bracket, the token pointer is not automatically advanced.

Another instance of the token pointer being automatically advanced is when a token matches a conditional. There are two types of conditionals: Relative and absolute. Relative conditionals use $=hh, $>hh, or $<hh. They should never be mixed with absolute conditionals, which use $hh=hh[,hh]. Absolute conditionals do not bump the token pointer (the first "hh" selects the token pointer explicitly). When "wrapup" encounters a relative conditional, it either matches it with the alt # of the current token, advances the pointer, and continues executing pattern elements following the last conditional in that sequence (only one conditional in a sequence needs to match), or it fails all the conditionals in that sequence and skips past the next ";" or "." element. If the element is a "." the token pointer is advanced. If the element is a ";" the next clause is executed. If the next clause does not begin with a conditional, the token pointer is not advanced.

The final case where the token pointer is automatically advanced is the OTRAN operation, XOR($). This operation takes the alt # of the current token and XORs it with the last EMIT byte. The rest of the time the + operation must be used to advance the pointer explicitly.

```
001A mark: XOR(40) hex .
```

Therefore, "mark" only gets invoked when an optional override is actually present. In this case, the previously emitted byte code 80, the "head" byte of the pattern, is XORed with the constant 40. This sets the flag bit in "head" that signals that an override byte follows it. Next, the "hex" pattern is invoked to translate the hex digits into a byte string and emit it into the object. "hex" calls the "b" primitive to build and EMIT each byte. In the PTRAN definition, the next pattern element is the "=" literal. Literals never produce a token, so this element is not mentioned in the OTRAN. Next, the PTRAN references "alt" which is defined as a bracketed sequence of "suc" patterns. When "alt" is matched, a token is produced giving the alt # of the "alt" that matched. Since "alt" has only one alternative, the token for it is ignored by the OTRAN definition, but it must nevertheless be skipped. That's why "mark" is followed by a + before calling "alt."

```
      001F alt:  suc :alt
```

Above, we consider the "wrapup" of "alt." This definition says to invoke "wrapup" of "suc" then continue "wrapup" at "alt." The construct ":alt" is a simple goto command. This would appear to set up an infinite loop. However, just as the design checks for an empty bracket and omits the "wrapup" of a matching subpattern, once a bracket is entered via a subpattern, a goto within that subpattern will not be executed if the token pointer is equal to the end of bracket pointer. A return from that level of "wrapup" is executed instead. This means that each trip through the "alt" pattern needs to advance the token pointer such that all tokens are matched just as the ":alt" goto is reached.

Now we come to the translation logic of each invocation of "suc" (above).

```
      0023 suc:  $=0     opt.prim;
      0027               $=1     prim .
      002B opt.prim:             EMIT(20) prim rep.prim end .
      0034 rep.prim:             prim :rep.prim
```

The PTRAN defining "suc" has two alternatives. By matching both cases with a conditional, we cause the token pointer to be advanced. The first alternative is a sequence of primitives inside repeat or option brackets. This is handled by the "opt.prim" pattern, which emits the "begin option" code of 20, followed by a "prim" and that by a "rep.prim." All of these elements utilize automatic token advancement.

```
      0038 prim: $=0     hex ;
      003C               $=1     g + ;
      0040               $=2     h + ;
      0044               $=3     i + ;
      0048               $=4     a + .
```

Here, primitive functions are called to handle each of the primitives, and the token pointer is advanced.

```
      004C end:  $=0     EMIT(20) ;
      0050               $=1     EMIT(7F) .
```

Following the bracket, the proper end of bracket code must be emitted: 20 signifies an option, 7F a repeat.

```
      0054 rep.alt:              < EMIT() sep + alt > c :rep.alt
      005F sep:  $=1     XOR(40) .
```

Finally, the original definition of "def" ends with **`> c rep.alt `** ` part of which is defined above. The reference to "rep.alt" is similar to what we've just been looking at, but we need to know how the rest of this definition works. The ">" operation pops an object pointer to the original "head" byte, and the primitive "c" inserts the difference between that index and the current "emit" index. This is the "skip" field within the "head" byte. A skip of 63 bytes is the maximum, anything more than that causes a semantic "overflow" error. Finally, at the end of the pattern is the ` primitive, which performs a final check on each statement.

Some Final Notes on PTRAN

PTRAN has three potential weaknesses in its design. The first is left recursion. If this is specified, but not prevented by the translator, a simple mistake will have been made that would cause the execution module to abort with a memory overflow. Left recursion is detected and flagged as an error. To replace left recursion, PTRAN has the [repeat].. construct. The second potential weakness is the effect of the order of alternatives. Notice the "end" pattern in the PTRAN definition of PTRAN. You might think the first alternative would always match, and the second alternative would never get attempted. After all, the second alternative begins with the first. PTRAN recognizes this source of errors within the alternatives of a single definition, and it reorders them so that they are matched in the correct sequence. It also renumbers the alternatives, so that their original sequence defines the alt #s assigned to tokens by "match." The third weakness is that PTRAN makes no attempt to detect or correct other ambiguities. It resolves ambiguity by selecting the first successful match in any pattern definition. PTRAN is able to translate a larger class of languages than most syntax driven translators are able to handle partly *because* of these weaknesses, and partly because of its ability to backup an arbitrary number of symbols during a match.

The final things yet to be explained are a couple of points about PTRAN alternatives. There are two ways to separate alternatives in a pattern definition. One is with a simple "|" (meaning "or"), and the other is with an "or equal" using the "|=" separator. This second form indicates that the alternative following it is to be given the same alt # as the previous alt. In fact, this is what the OTRAN "sep" pattern does; it marks the "next" byte using an XOR(40) operation if the matching alt # is one. Finally, you may have noticed that the definition of an "alt" is zero or more "suc" patterns. This means that an "alt" may be empty. An empty "alt" (or pattern) always fails. It always forces the next match alternative to be attempted. It allows an easy and visual way to indicate alt #s that aren't to be generated by a match. An empty alt essentially says, "add one to the alt # and match the next alternative."

The OTRAN Backend

The OTRAN design permits two different kinds of translators. One type is for languages that have variable length statements and use options and repeats, or right recursion. These languages need relative references to the tokens. The other type is for languages with short statements that have 15 or fewer, and fixed sequences of tokens. These languages may employ absolute conditionals to reference the tokens. The following table shows with binary numbers the encoding and interpretation by "wrapup" of the OTRAN object data structure.

```
0000 0000                   . end
0000 0yyy [byte]..   emit the yyy (1-6) bytes that follow
0000 0111            emit a 0 byte
0000 1000            ; end
0000 1001            < action
0000 1010            - action
0000 1011            > action
0000 1100            + action
0000 1101 (byte)            XOR with last emitted byte
0000 1110 (byte)            GOTO :label
0000 1111 (byte)          * CALL WRAPUP(label) a subpattern reference
00xx yyyy (byte)[byte]      $b=b[,b] absolute conditional, yyyy = 1st
b
0011 yyyy          * $<b relative conditional, yyyy = b
0011 1111 (byte)          * $<b relative conditional, byte = b
0100 yyyy          * $=b relative conditional, yyyy = b
0100 1111 (byte)          * $=b relative conditional, byte = b
0101 yyyy          * $>b relative conditional, yyyy = b
0101 1111 (byte)          * $>b relative conditional, byte = b
011 yyyyy            CALL primitive = yyyyy (0 - 31)
1xxx yyyy            XOR($b,b) Lshift $yyyy, xxx bits & XOR
1xxx 1111          * XOR($) Lshift current alt, xxx bits & XOR
                            (target of above two is last emitted byte)
                          * = Automatic + action on token pointer
```

The syntax of OTRAN allows great freedom in generating the above data structure. In actual fact, however, an OTRAN specification must very closely follow the language of the intermediate tokens produced by the front-end of a translator. Making this correspondence is up to the engineer. You need to know exactly what token sequences can be produced by the PTRAN front-end, so that the OTRAN backend will exactly match them. This means that every call to one of your primitive functions must have the token pointer just where you expect it to be.

The whole purpose of both PTRAN and OTRAN is to handle as many aspects of the translation process as possible, and make the primitives that you have to code as simple and detached from one another as they can be. Although PTRAN and OTRAN are capable of being used in almost any text to binary translator, their purpose here is to support XPL development. For this reason, once they had bootstrapped themselves, they were used to produce the DEXPL translator. This is where we will turn our attention next.

DEXPL - A Translator to Define XPL

The word DEXPL will be used like the words PTRAN and OTRAN — to name both the language and its translator. The DEXPL language is a metalanguage for describing different XPL assemblers. Given the basic XPL program linked to the .OBJ file produced by DEXPL, you have an assembler that, although able to run only in a DOS window on an Intel processor, is able to generate object code for almost any machine that exists. The power to accomplish this is contained in the DEXPL specification language. This language is defined in PTRAN, and it will be in PTRAN that we will learn about it. First, however, it might be best if the XPL language itself were explained. This will be done in more familiar terms than the PTRAN metalanguage. XPL (those aspects of it that are common from one DEXPL-specified dialect to another) will be described in words and through examples.

XPL - the Language in General

An XPL source file may contain one line or over a hundred pages. Each .SRC (source file) produces a single .OBJ file that LINK can combine with zero or more other .OBJ files into a .COM file. A .COM file is nothing more than an absolute binary file that needs no modification to be run on a given processor. Simple utilities can be written to change .COM files into .EXE files, or combine them into whatever type of file is executable under a given set of circumstances.

A line of XPL text can have one of six forms: The line is a PRINT command if a "." appears in column 1. The line is a comment if a ":" appears before column 7, or if the line has fewer than 6 characters, or if the line begins with "<<" (in which case, the comment continues through a line that contains the characters ">>"). The line is a directive if an "=" appears in column 5 or 6. The final three types are: An XPL declaration, a line of XPL code, or a "suffix line" added by XPL to the end of a source file to list all external symbols and their address of initial reference (this case is matched by the final alternative of the definition below).

All lines are initially matched with the "prep" definition. The "prep" definition is prepared with PTRAN, and its corresponding "back" definition is prepared with OTRAN. All "body" definitions and their corresponding "back" definitions are prepared using DEXPL. Only if the "prep" match fails is the line matched with DEXPL patterns. The "prep" definition and all of the program code do not change from one version of XPL to another. Only the DEXPL definition changes. Each dialect or different target machine is completely specified using only a different DEXPL source file. This produces a single .OJB file that is linked with all the other XPL object files to produce a working translator for that unique assembly language. Another aspect of XPL that is the same across all of its possible dialects is the unique block structure of XPL.

XPL - Its Block Structure

XPL is a block structured language, like no other block structured language. It is a programmer's language, not a compiler writer's language. It's a little harder to translate, but it's easier to read and write than any other assembly language. You have already seen examples of XPL's block structure. Now, you will learn how it's defined.

XPL's block structure is defined using the context of its indentation and four types of reference keywords. Each line of XPL code is defined (in DEXPL) as a >prefix, NEXT, or EXIT line. EXIT is the default, and forms the only possibility in most other languages. First, let's be clear about the difference between a statement and a reference. A statement is a complete definition in DEXPL. A reference is a special construct within a statement. Explicit DEXPL syntax is necessary to declare a statement to be a >prefix or a NEXT statement. Otherwise a statement is an EXIT statement. An XPL definition (in DEXPL) allows code addresses to be referenced either explicitly or implicitly. A statement such as "GOTO <label>" is an example of an explicit reference. XPL block structure, on the other hand, makes use of implicit references. An implicit reference makes use of the block structure context.

The block structure context is a combination of the indentation and the way it can be referenced. This is done with the reference prefix symbol, and the loop, next, and exit references. Let's continue this explanation with an example.

```
A001          DO    A=A+1
A002          *     IF ZR,EXIT
A003                CALL ZIP
A004                IF CY,
A005                      CALL ZOT
A006                *     IF NC,LOOP
A007          LOOP
```

I have no idea what this block of code might do, but I can see all the branching possibilities at a glance, and I hope you can, too. The first line is an example of a >prefix statement (the DO keyword) followed by a statement that adds one to “A.” Line 2 begins with a reference prefix, its “body” is indented, and it contains the keyword EXIT (which happens to be associated with a DEXPL exit reference). A reference prefix designates the indentation level referred to by any loop, next, or exit reference contained in that statement. If no reference prefix is required by the statement, it is illegal to use one. If one is required, but none is present, the default prefix is where the body itself begins. In the case of line 2, a branch is taken past the LOOP statement if the ZR condition is true. We’ll see why the branch is to the line following LOOP, and not directly to LOOP, in a moment.

Lines 3 and 5 were inserted to give some possibility that this block of code might actually do something. Line 4 is the next line that will interest us here. It indicates that if the CY condition was returned from ZIP, the call to ZOT is to be executed. Otherwise, branch to the LOOP instruction (not past it, but directly to it this time). This is done in DEXPL by building a “next” reference into the IF statement. In line 6 another conditional branch follows the call to ZOT. It uses the LOOP keyword, and as you might expect, is associated in DEXPL with a loop reference. As you can see, lines 6 and 7 both employ LOOP keywords. However, line 6 branches to the line that follows the IF statement to which its reference prefix refers. Line 7 branches to the DO statement. These choices are made based on the type of statement that the loop refers to. If a “loop” references a NEXT statement, its target is the statement following the NEXT statement. This distinction also explains the difference between the branches taken in lines 2 and 4. Line 2 contains an “exit” reference to a NEXT statement, so it branches past any NEXT statements to the first EXIT statement. Line 4 contains a “next” reference, so it branches to the first NEXT statement that follows it. If an EXIT statement is referred to by a “next” reference, a branch will be taken to it by default.

Remember, a “loop” reference to an EXIT statement targets that statement. A “loop” reference to a NEXT statement targets the first statement following it. A “next” reference targets the first statement encountered, be it a NEXT or an EXIT statement. An “exit” reference targets the first EXIT statement encountered, skipping any intervening NEXT statements. All of these occur at the indentation levels of their reference prefixes. Each line that changes indentation, causes processing that handles block structure references. A final word about the example: You may have noticed a better way to code it, but I wanted to show as many possibilities as I could. In real life, line A004 would be * IF NC, LOOP, and line A005 would be DO CALL ZOT, followed by LOOP WHILE NC.

DEXPL Primitives

Unlike PTRAN and OTRAN, all primitives in DEXPL, other than literals and subpatterns, are defined in advance. There are only 12 available to the DEXPL programmer, although there are a plethora of special controls in the language. The 12 primitives that may appear in DEXPL source are defined as follows:

```
   .#        = symbol [addop symbol]         a byte, ordinal
   +#        = addop symbol [addop symbol]   a byte, integer
   -#        = [addop] symbol [addop symbol] a byte integer or ordinal
   ~#        = symbol [addop symbol]         a self-relative byte
   .#@       = symbol [addop symbol]         a word, ordinal
   +#@       = addop symbol [addop symbol]   a word, integer
   -#@       = [addop] symbol [addop symbol] a word integer or ordinal
   ~#@       = symbol [addop symbol]         a self-relative word
   .##       = symbol [addop symbol]         a quad, ordinal
   +##       = addop symbol [addop symbol]   a quad, integer
   -##       = [addop] symbol [addop symbol] a quad integer or ordinal
   ~##       = symbol [addop symbol]         a self-relative quad
  where,
   symbol    = qhex | biny | [pfx] u 4 | u 5 | [pfx] vhex |
               reg u 6 | [pfx] u 7
   addop     = + | -
   reg       = AA|BC|DE|HL|SP|BP|SI|DI|A|B|C|D|E|H|L
   qhex      = "'" hd hd [hd hd] "'" | "'" hd hd hd hd hd hd hd hd "'"
   biny      = bd bd bd bd bd bd bd bd [bd bd bd bd bd bd bd bd] u 8
   vhex      = 0 hd [hd] [hd] [hd] [hd] [hd] [hd] [hd] |
               cd [hd] [hd] [hd] [hd] [hd] [hd] [hd]
   bd        = 0 | 1
   cd        = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
   hd = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F
   pfx       = | | @ | #
   << The following "u" primitives complete the above definition:
   u 4 = match and reference a [~|`] <decimal> literal
   u 5 = match and reference a byte/word/quad "string" | ^byte
```

```
        u 6 = fail if next source is any alpha.num, else call SET.FAIL
        u 7 = match and reference a <label>
        u 8 = fail if next source is any <hd> character
        >>
```

Order in the above is extremely important. It resolves the ambiguity that would otherwise exist between binary, decimal, and hex literals. Binary consists of exactly 8 or 16 binary digits. If this isn't the case, decimal is matched next. It consists of variable length strings of decimal digits (without a leading 0 if more than one digit, and not followed by any <hd>). If this doesn't match, hex is tried. It must have a leading 0, or contain one of the special hex digits, otherwise it would have matched the "binary" or "decimal" patterns.

The "pfx" characters provide a way to force a smaller value to be interpreted as a word or quad value. The ` character indicates that the most significant bit of the value is to be set to one. The ~ character indicates the binary complement of the value that follows. The ^ character indicates a value from 00 to 1F, and is followed by an ASCII character in the range 040 – 05F (@ A … Z [\ ] ^ _).

Two symbols are treated differently when the target machine is Big Endian. These are Hex and ASCII string symbols. They are byte reversed when stored in the symbol table, so that when they are stored into the code and a second byte reversal takes place, the string will be in its proper sequence.

When a primitive is matched, its OTRAN counterpart is also matched, and the final result is passed back to the DEXPL "body" being matched. The evaluation of a primitive results in one of three outcomes. It may fail syntactically. Or, it may fail based on its value. Quad values don't match word primitives, nor do word values match bytes. Or, its value may be indeterminate, and it may be accepted provisionally. Provisional acceptance causes one of two results. If the value is later determined within the source file and is inappropriate to the primitive, another pass is performed. That statement is then rematched with the value now being known (approximately), and will presumably have different results. Again, order is important. If a quad primitive is matched before a byte primitive is attempted, it will always match and the alternative containing the byte primitive will never be tried. The second result of a provisional match is that a label is not defined within the local source, but is only resolved later by LINK. In this case, the programmer has to perform the backup, by using an explicit "pfx" (an @ or # in front of the label) to avoid the LINK error.

DEXPL Definitions

The format of a DEXPL statement is similar to that of PTRAN and OTRAN. Ignored lines are exactly the same. It has two meta-statements:

```
        NAME= f e
```

Defined exactly as for PTRAN and OTRAN, if this statement is used, it may only be preceded by ignored lines.

```
        BASE= hex [; parm].. e
```

This statement allows up to three things to be defined with "parm." The default value size is given by the keywords 16 BIT or 32 BIT. The type of values are given by LITTLE ENDIAN or BIG ENDIAN. And, a version may be given by V.0, V.1, V.2, or V.3. The defaults are 16 BIT, LITTLE ENDIAN, and V.3. The values defined in this statement are passed along to XPL in the form of two external symbols called BASEWORD (from the "hex" pattern) and PARMBYTE (the upper four its being defined by "parm" equates, with the defaults being 1 bits).

Following the above (if any), a DEXPL source file has two further types of statements: Pattern definitions and source-object definitions. Pattern definitions are a simpler version of PTRAN pattern definitions. They are defined in PTRAN as follows:

```
        pattern    = a [d] = alt [sep c alt]..
        alt        = [prim]..
        sep        = "|" | "|="
        << where c allows continuation and d matches 2+ spaces>>
```

Source-object definitions have a variety of forms all stemming from the simple pattern:

```
                     source [m object] e
        where,
        source     = [pfx.0] d line [d specs] [m src]..
        src        = [pfx.1] d line [d specs]
        object     = [flg] d emit [emit].. [m obj].. [:a]
        obj        = [a:] d (ref = bb) [d] [emit].. |
                     [a:] d emit [emit]..
        bb         = b | b,b
        << where, pfx.0, pfx.1, line, specs, flg, emit, and ref are defined
```

```
on page 18.  a = same as on p.10; b = binary; m = continuation.>>
```

The primitive “a” is the same alpha.numeric label as in PTRAN and OTRAN. The primitive “b” is a binary number of up to 8 binary digits. The primitive “m” signifies mandatory continuation (the current line must end and the next line continues the current DEXPL definition). Typically, “m” begins an option, so if the next line does not continue the current definition, the option fails and the match continues from there.

In words, what is defined above is one or more “source” lines followed by zero or more “object” lines. The “source” lines define alternate bodies for XPL statements that share a common “object” definition. “Source” lines can begin with optional prefix characters and end with optional “specs” (side effects or conditions). “Object” lines can begin with optional flags or labels. And, their “emit” lists may be preceded by a “ref” (conditional). Let’s learn from examples. Here is a pattern and a source-object definition that references it:

```
A000 reg = A | B | C | D | E | H | L
A001                 reg = reg' + 1
A002                 [00101 $1]
```

When this definition encounters the source, “E=E+1” it will cause the byte 00101100 to be emitted. If the source, “E=B+1” is matched, it will fail. Why? The first occurrence of “reg” matches any of the alternatives listed. It also defines the token pointer at the point of match because of the side effect of the ' following the second “reg.” This modifies the pattern to contain an override byte before the first “reg” and the rematch primitive in place of the second “reg.” The PTRAN for this was discussed earlier. Each subpattern reference or primitive in a “source” line defines a fixed token. Tokens are referenced by the symbols $0, $1, … $D, $E. A maximum of 15 tokens may be referenced. In the above definition, $1 represents whatever alternative is matched by “reg.” $0 represents the statement level alternative itself. Now, let’s take a more complicated example using the same “reg” pattern.

```
A000                 reg = $ .#@
A001                 $ .#@ = reg
A002                 [1100 $0 $1] $2
```

Now, what’s up? Here, there are two alternatives at the statement level. The first follows “reg” with two literal characters (=$), then by a primitive (a word ordinal, defined as “symbol [addop symbol]”). The alt # of the “reg” that matches will define the $1 token and the word value will define the $2 token. However, in the second “source” alternative, these two tokens are reversed. The DEXPL translator recognizes this possibility and attempts to order each alternative like the first, by inserting token override bytes. Three bytes of object are produced when this definition is matched: The byte code indicated in brackets, and the word value indicated by $2. When DEXPL detects an inconsistent token reference, or an ordering it can’t resolve, it will flag an error.

There are still lots more things we might like to do when defining an assembly language, so we need ways to say them in DEXPL. For example, certain instructions set various combinations of condition codes. It’s nice to have XPL flag these instructions in the XPL source. To handle this, DEXPL allows the first line of the “object” to have a flag prefix character. These include any of the characters ! ‘ “ # $ & * + - and . Using the * flag, the following DEXPL lines might be defined (I’m just making this up):

```
B000                 reg : reg
B001  *              [10 reg reg]
```

This would put an * in column 5 of every line of source that matched this pattern. At this point, you may have guessed that each line of a DEXPL “source” pattern corresponds to one alternative of a PTRAN definition. And, you would be right. However, DEXPL normally tacks on a “p” (end of line) primitive to every “source” line. This saves writing them explicitly and makes the definition cleaner, but sometimes you want to define a statement that is nothing more than a prefix to another statement, so you want the next “source” pattern match to continue on the same line. Two examples follow:

```
C001  >              DO

C002 sreg = CS | DS | ES | SS
C003  >              sreg:
C004                 [001 $1 110]
```

On line 1, the DO keyword is defined as a prefix. Any statement may follow on the same line and, by itself, DO generates no code, but it does define DO as an EXIT statement (by default). Line 2 defines the “sreg” (segment register) pattern. Lines 3 and 4 reference this pattern and define the four codes, called segment override instructions, that are typically followed by memory references that are modified by these prefix codes.

Sometimes, the people who design computers get enthusiastic and define instructions that do several things at once. Intel did this with some of their string instructions. They also have separate instructions that can do single steps of these more complex operations. So, if you modify the following sequence in any way, different definitions generate other instructions. If you follow it exactly, only a single instruction is generated.

```
D001  <           DO UNTIL BC=0
D002  <           s.prim
D003  <           BC=BC-1
   D004  <        LOOP NE (BC>0)
   D005           LOOP EQ (BC>0)
   D006  =        LOOP (BC>0)
   D007           [1111001 $5] [1010 $3]
```

The above definition works as follows. Each of the < characters signifies that the first alternative of a multi-line definition is given on that line. Line 1 must be followed by lines two, three, and so on (on separate lines of source), in order to match. I haven't shown the pattern for "s.prim" because it is too complex for my purposes here. Suffice it to say that it represents a string primitive operation that could also generate a stand alone instruction. Any one of the lines 4, 5, and 6 may follow line 3. They are alternatives, and line 6 is an "|=" alternative (it is given the same alt # as line 5), as indicated by the "=" prefix. DEXPL has to create four subpatterns to represent the above, and that means that $0 will always contain a 0, since there is only one alternative in the overall pattern (containing references to the four subpatterns, one after another). Tokens $1 and $2 will also always be 0, since they also have only one alternative (DO UNTIL and "s.prim"). The next token, $3 is set according to the alternative of "s.prim" that matches. This is the token that we plug into the second variable field of the instruction code. Next, $4 is always 0, because it corresponds to line 3, and $5 is defined by the choice made between lines 4 through 6. This is really a binary choice, since the third alternative is equivalent to the second. Therefore, it describes a 1 bit field that replaces the $5 reference in line 7. It turns out that four of the "s.prim" instructions cause condition codes to be set, so it would be nice to have a way to add this information to the above definition, and have XPL flag those codes, but leave the flag field blank for all the rest. This is done by using one of the options of the "specs" pattern. We would rewrite line 2 as follows:

```
D002  <           s.prim  ($3=0110,0111 *),($3=1110,1111 *)
```

These "specs" check the alt # in $3 and emit the * flag (into column 5 of the source) if the alt # is equal to any of the binary values listed. As always, the "s.prim" pattern has to be defined in the proper sequence so that its alt #s are in exact correspondence with the binary codes they represent.

Now it's time to define a few of the patterns we promised to define on page 16.

```
E001 pfx.0 = < | >
E002 pfx.1 = < | > | '='
E003 line  = prim [prim.i]..
E004 specs = spec [, spec]..
E005 flg         = | ! | '"' | # | $ || & | "'" ||| * | + || - | .
E006 emit  = ref | '[' field [field].. ']' | ! b
E007 ref         = $0 | $1 | $2 | $3 | $4 | $5 | $6 | $7 |
E008               $8 | $9 | $A | $B | $C | $D | $E
E009 prim  = .# | +# | -# | ~# | .#@ | +#@ | -#@ | ~#@ |
E010               .## | +## | -## | ~## | g | l | a
E011 prim.i      = prim [i]
E012 field = b | ref
<<   a - (defined earlier) is a subpattern reference
     g - matches a quoted string (a literal keyword)
     i - matches an apostrophe ' character (wrapup checks for this,
            and inserts the override and the rematch primitive)
     l - matches any literal except x#, alpha, control, space, or | [
>>
```

The prefix characters in lines 1 and 2 have already been discussed. In the third pattern, each "line" is nothing more than a sequence of "prim" patterns. The second or a subsequent "prim" may be prim' (with an apostrophe) to indicate a rematch of the "prim" of the same name that must have occurred earlier on the same line (otherwise "wrapup" flags an error). We have seen an example of a "spec" but we will discuss the full set of options below. We have also seen an example of the "flg" prefix. Above, are the full set of "flg" characters (there are 10). The "flg" pattern is defined with empty options to allow the alt # to be equal to the lower four bits of the ASCII code for the character itself. Remember that tokens are not generated when a literal is matched, so the alt # is all we have to go by. Finally, there are three alternatives in the "emit" pattern: A simple "ref," one or more "field" patterns enclosed in brackets (here, brackets do not signify that the emit is somehow optional, they merely group the fields of a single binary byte), and the "! b" pattern. We have seen examples of the first two: They cause a byte, word, or quad value to be emitted (a "ref" outside of brackets), or a single byte composed of "fields" to be emitted. The third alternative indicates that the previous byte value emitted should be XORed with the "b" constant. This has the effect of flipping selected bits of the last byte EMITted, and it is normally appended to a bracketed fields definition. For example, the following definition emits two instructions: A test conditional, and a RETURN. Since the RETURN is to occur if the condition is true, the branch must be taken when it is false. The !1 flips the least significant bit of the condition and reverses its sense (it's this simple on Intel, other machines could vary).

```
        F001              IF cond, RTN
        F002              [0111 $1]!1 [00000001] [11000011]
```

Here's an example of the above that leads us on to a fuller discussion of the "spec" pattern.

```
        H001              LOOP            NEXT,$1=LOOP,REL,BYTE
        H002  =           AGAIN           $1=LOOP,REL,BYTE
        H003  =           NEXT            $1=LOOP,REL,BYTE
        H004  =           EXIT            $1=LOOP,REL,BYTE
        H005              LOOP            NEXT,$1=LOOP,REL,WORD
        H006  =           AGAIN           $1=LOOP,REL,WORD
        H007  =           NEXT            $1=LOOP,REL,WORD
        H008  =           EXIT            $1=LOOP,REL,WORD
        H009              [111010 $0 1]!10 $1
```

This shows the definition of eight unconditional block structure branch instructions, the first four are short branches, specified by a byte value, the second four are longer branches, specified by a word value. The first four are all equated (forming alt 0, hence setting $0 to 0), and the second four all set alt 0 to 1. Since alt 0 is used to define a one bit field, and the sense must be reversed, we need the !10 to flip the particular bit. Now, you wonder, why didn't we just reverse the order of the alternatives, so they would define the field correctly in the first place? The reason is that order is important. The first four alternatives have "specs" that define $1 as a byte value. If the second four alternatives preceded them, a word value would be requested, and a byte value would always match. If the byte values came after the word values, they could never be matched. Here is the full definition of "spec."

```
        I001 spec  = except | NEXT | GLOBAL | FPFX | def
        I002 except        = < ref = bb [& ref = bb] >
        I003 def           = ref = b | ref = mod [,mod] [,mod] | (ref = bb flg)
        I004 mod = LOOP | NEXT | EXIT | ORD | INT | REL | BYTE | WORD | QUAD
```

Notice the keyword NEXT in the previous example. It is customary to put the NEXT keyword to the left of the "def" construct. It would technically be correct to put it after the $ref list, but it will be taken as part of the list unless three keywords precede it, and it would not take effect until after the $ref were processed (which could alter the result).

Now, let's take the above definition one element at a time. The first type of "spec" is an "except" (exception). This spec is enclosed in < > brackets. It allows one or two "ref=bb" patterns. This construct tests the value of the given token against one or two literal binary bytes. If equal, the exception is true, so the match of this alternative fails. Several exceptions may be tested. If any are true, the alternative fails. When two tests are connected into a single exception using the & option, both have to be true for the exception to be triggered.

The next "spec" is the NEXT keyword. Be sure it isn't positioned so as to be confused with a NEXT "mod." This keyword designates a statement matching the current alternative as a NEXT statement for block structure purposes. Remember that the default, if NEXT isn't present, is that the statement is an EXIT statement.

The next "spec" is the GLOBAL keyword. Two of the attributes of a label are LOCAL and GLOBAL. A local label (the default, so we need no keyword to designate it) is visible only inside the current source module. If LINK finds an external reference to it, a global label is visible from another module. Within a module, the same label may not be used for both purposes. Between all the modules to be LINKed, a GLOBAL label must not be defined more than once. Any label that is referenced, but not defined, in a module has a third attribute called EXTERNAL. This label must be defined as a global in some other module.

The FPFX keyword is the final "spec" keyword. It requests the definition to fail if a * (reference prefix) is present on the statement.

The final "spec" is the "def" pattern, with its three alternatives. The first alternative defines the value of a "ref" as an absolute binary byte. The second alternative defines the value of a "ref" in terms of the LOOP, NEXT, and EXIT keywords (using the statement's reference prefix and the prevailing block structure), or, if these are missing, it places restrictions on the "ref" value, such as INT (integer) or REL (self relative). Or, it defines the actual size of the "ref" value as a BYTE, WORD, or QUAD. Or, it can do any combination of the above. Choose at most one keyword from each sequence of three to define each "mod." REL and BYTE are the defaults.

The last alternative of "def" is one we've already seen. It tests the value of "ref" and if it matches any of the binary bytes listed, a flag character is written into column 5 of the current line of source.

In summary, the purpose of DEXPL is to associate the definitions of source statements with the binary object that executes those statements. To accomplish this, DEXPL allows you to define the machine environment, a set of patterns to be used in subsequent definitions, and then alternative source lines that generate a particular sequence of object code. Both single instructions and “macro” combinations can be defined. Any byte, word, or quad oriented architecture can be targeted. Both little and big endian machines are addressed, however only one line of machines may serve as the host: An Intel 8086 or compatible (through the Pentium and beyond) running PC-DOS.

Data Structures

We have already seen the PTRAN pattern data structure. It is a byte string interpreted one pattern alternative at a time, with some context dependencies. Later, we studied the intermediate token sequence output by “match” and input by “wrapup.” All of the programs discussed in this document use these two pattern structures and the token intermediate (PTRAN, OTRAN, DEXPL, and XPL). However, XPL uses a more extensive form of the object structure than do any of the others. PTRAN, OTRAN, and DEXPL use a minimal .OBJ definition. Essentially, this is the common subset between the “old” format used by the XPL and LINK that supported the development of all these tools, and the “new” format that will be used by the XPL and LINK that will result from this project.

The .OBJ Data Structure

During translation, all four of the translators discussed here keep a symbol table and object output. The .OBJ file is simply both of these together. The format of the .OBJ file will be discussed in three parts: Minimal .OBJ files, Old .OBJ files, and New .OBJ files.

Minimal .OBJ Files

The symbol table part of the file consists of the following sequence of bytes:

```
        <0FB> <0> <0> <NAME=filename, 8 characters w/blank fill>
        [<00xx skip> <@value> <symbol>].. <0> <entry count>
```

The first byte is the signature byte. It indicates “old type” .OBJ file, defined as 16 bit words, little endian, and version V.3. The next two bytes are zero. The NAME filename is padded to eight characters with spaces. The skip field 0B (11 decimal) reflects the entry’s 11 byte length. The skip fields in subsequent entries vary from 4 to 13 (decimal). These entries contain the GLOBAL symbols declared by the module. These symbols may have any of three types of definition, depending on the value of “xx” in their head bytes. If “xx” = 00, the “@value” (word, or two bytes), is the absolute value of the symbol. If “xx”=10, a code relative value is declared. And, if “xx”=11, the value is data relative. The final two bytes are a zero byte, followed by a count of all the entries in the table. This bit of redundancy sorts out a non .OBJ file from a real one. The second part of a .OBJ file, called the fixup section, contains another <skip> <string> repeated sequence.

```
        <00 skip> <data used> <absolute code>
        [<xx skip> <bytes to be fixed> <absolute code>].. <0> <count MOD 256>
```

The lower six bits in each skip byte are a skip from one to 63. The final two bytes are a zero as an end skip, and a count of the number of skips. The “data used” is a word value giving the byte count of data allocated to the module. An empty .OBJ file, named FILE, would contain the following sequence of bytes & words.

```
                        = 0FB,@0,"FILE    ",0,1
                        = 03,@0,0,1
```

The size of each skip depends upon how many bytes of absolute code there are before the first or next fixup. Each skip goes to the next fixup until the final zero byte is reached. The “xx” bits of skips beyond the first are as follows: 00 means no fixup (the last skip was probably 63, and another skip had to be generated), 10 means the next two bytes are code relative, and fixup consists of adding the code base for the module to them, 11 means the next two bytes are data relative, and fixup consists of adding the data base for the module to them. Bits “xx” = 01 are not used in the minimal .OBJ file. Only the final entry may have a skip = 0, and its upper bits must be zero as well.

The Pre-98 .OBJ File Format

The symbol part of the file is similar to the above with three additions. First, the two bytes after the signature entry are as follows: the first is the final word token allocated, the second is the final byte token allocated. Word tokens increase from 1 to n. Byte tokens decrease from –1 to –m, n+m being a maximum of 255. Second, there may be additional 0FB entries after the first, naming LOAD= modules (the two bytes between skip and filename, being meaningless in their case). And third, if the “xx” code in the skip byte is 01, the entry names an external label that must be defined by LINK, and all its references must be fixed up. In this case, a word token is recorded in the first byte position, and a byte token in the second position. Other codes that may appear are:

```
        <1000 0100> <token> <word value>  -  token = absolute value
```

```
<1001 0100> <token> <token>< X >  -  1st token = 2nd token
<1010 0100> <token> <word value>  -  token = code relative value
<1011 0100> <token> <word value>  -  token = data relative value
<1110 skip> <token> <expr> <word value> <optional word>
<< in the last of these, token = <uu>(<word1> <bb> <word2>)
and, expr = <w1 w2 bb uu>, and skip = 0101 or 0111, where:
w1 = 00 absolute word      uu = 00 no op      bb = 00 2nd word missing
 & = 01 external word         = 01 -(w1 bb w2)   = 10 (w1 + w2)
w2 = 10 code rel word         = 10 ~(w1 bb w2)   = 11 (w1 - w2)
   = 11 data rel word
>>
```

The fixup section of an old .OBJ file also has some additions over the minimal .OBJ file. As you can imagine, in both sections, these additions are not compatible with the new .OBJ files. The upper two bits in the first skip are interpreted as follows:

```
<00 skip> <DataUsed(2)> <absolute bytes>
<01 skip> <DataUsed(2)> <OrgData(2)> <absolute bytes>
<10 skip> <DataUsed(2)> <OrgCode(2)> <absolute bytes>
<11 skip> <DataUsed(2)> <OrgCode(2)> <OrgData(2)> <absolute bytes>
```

Subsequent skip bytes are interpreted as follows:

```
<00 skip> <absolute bytes, no fixup>
<01 skip> <token> <nil|0|1> <absolute bytes>
<10 skip> <code relative fixup> <absolute bytes>
<10 skip> <data relative fixup> <absolute bytes>
<<if <token> is a byte token, the next byte is nil (not present)
  if <token> is a word token, and
           the next byte = 0, the fixup is an absolute value, or
           the next byte = 1, the fixup is a self-relative value.
>>
```

Old .OBJ files only allowed an absolute CODE= or DATA= declaration once at the beginning of a module. Once the code and data base addresses were assigned, all code and data relative addresses within the module were offset by those values. Modules that declared no CODE= value were listed as beginning at 0000 and relocated to whatever the next available byte address was as determined by LINK.

The Post-98 .OBJ File Format

The new .OBJ file format is identified by its first byte, a signature byte, whose skip field is 2-9 as opposed to the fixed 0FB of the Pre-98 format. The old LINK couldn't handle a collection of .OBJ files as large as 64K, but the new LINK will be able to handle any set of .OBJ files that result in no more than 64K of linked absolute. Each of the new .OBJ files may have sections as large as 64K. The complete definition of the new format follows.

The .OBJ Symbol Table

```
<abcd skip> <filename(1..8)>     Signature Byte (skip = 2-9) <root filename>
      ab = Version Number (00-11 = V.0 thru V.3) 11 = default or Minimal
      file.
       c = 1 means Values are Little Endian (default or Minimal file)
           0 means Values are Big Endian
       d = 1 means Values are Words (default or Minimal file)
           0 means Values are Quads
```

The complete first entry in the .OBJ file comes from the NAME=filename. If this is absent, the source filename is used by default. Following the signature byte is a 1-8 character filename. A null or empty source file (that has no errors) will produce a null .OBJ file like the one indicated above (see minimal .OBJ file), except the signature may be different, and there will be no blank fill in the filename entry. A source file with errors will cause its corresponding .OBJ file to be erased. However, a translation that is aborted for some reason will change neither the source file, nor erase the .OBJ file. The purpose of the signature byte is to select an option built into LINK for up to 17 versions of object file (including the minimal file definition, but not the extended pre-98 definitions).

Following the signature entry, .OBJ files may have a number of additional entries that follow the form:

```
<0Fx> <filename(1..8)>
where, x=2..9 (the upper bits, all 1's, indicate a file symbol entry)
           and x = the entry's total length, a byte skip count.
```

These entries come from LOAD= declarations (or additional names on a NAME= statement). They inform LINK of additional files it must access to be linked to the current file. The root file of an entire collection is the file from which the final .COM file is named. The initial 0Fx entries in a symbol table are not accessed for normal lookup purposes. No 0Fx entries, following the first non-0Fx entry, are produced by XPL.

Symbol table entries following the Root and Load entries are a string involving three parts: The type-skip byte, the value-token word (or quad, if a value is present and the "d" bit of the signature = 0), and a variable length name, string, or expression. The format of each type of entry is determined from the type field of the type-skip byte:

```
0x <value(x)> <name(1..10)>        not produced by XPL
1x <value(x)> <name(1..10)>        not produced by XPL
2x <value(x)> <name(1..10)>        not produced by XPL
3x <value(x)> <name(1..10)>        not produced by XPL
4x <xx:Token> <name(1..10)> External/Forward Ref (Mark)
5x <xx:Token> <name(1..10)>        not produced by XPL
6x <00:Token> <name(1..10)> LOCAL, 2nd entry = value def (Dx,E5)
7x <00:Token> <name(1..10)> GLOBAL, 2nd entry = value def (Dx,E5)
8x <xx:Token> <name(1..10)>        not produced by XPL
9x <xx:Token> <name(1..10)>        not produced by XPL
Ax <KeyToken> <name(1..10)>        not produced by XPL
Bx <KeyToken> <name(1..10)>        not produced by XPL
Cx <xx:Token> <string(1..12)>      not produced by XPL
D4 <KeyToken> <value(1)>           Ordinal Value < 256
D5 <KeyToken> <value(2)>           Ordinal Value < 65536
D6 <KeyToken> <00:Token> <OpKey(1)>                t0 = u t1
D7 <KeyToken> <value(4)>           Ordinal Value < 4294967296
D8 <KeyToken> <00:Token> <OpKey(1)> <00:Token>     t0 = u t1 op t2
E5 <KeyToken> <00:Token>   token equate    t0 = t1 (Dx)
Key = high 4 bits of KeyToken as follows:
0 - (context)      4 - OrgCode ABS 8 - OrgCode CR  C - OrgCode DR
1 - Byte ORD       5 - Byte INT    9 - Block Ref   D - Def Token
2 - Word ORD       6 - Word INT    A - Word CR     E - Word DR
3 - Quad ORD       7 - Quad INT    B - Quad CR     F - Quad DR

           CR = Code Relative      DR = Data Relative
Key=4,8,9,C,D are Refs in XPATCH; 1,2,3,5,6,7,A,B,E,F are Def/Ref.
Key=0 in XPATCH is an OrgCode to an External.
Key>0 (Def) in all type Ax and Bx entries.

OpKey =    01110000 unary Self-Relative    1xxx0011 unsigned /
           0xxxxxxx No binary Op           1xxx0100 unsigned \ (MOD)
           x001xxxx unary ~ (ORD,CMP)      1xxx0101 signed `\ (`MOD)
           x010xxxx unary + (INT,NOP)      1xxx0110 signed `*
           x011xxxx unary - (INT,NEG)      1xxx0111 signed `/
           1xxx0000 binary +               1xxx1000 boolean AND
           1xxx0001 binary -               1xxx1001 boolean OR
           1xxx0010 unsigned *             1xxx1010 boolean XOR
```

The .OBJ Fixup Area

The fixup section of a new .OBJ file is only a slight modification of the old format. The upper two bits in the first skip are interpreted as follows:

```
        <00 skip> <DataUsed(2)> <absolute bytes>
        <01 skip> <DataUsed(2)> <CodeMask(2)> <absolute bytes>
        <10 skip> <DataUsed(4)> <absolute bytes>
        <11 skip> <DataUsed(4)> <CodeMask(2)> <absolute bytes>
```

Subsequent skip bytes are interpreted as follows:

```
        <00 skip> <absolute bytes, no fixup>
        <01 skip> <token(2)> <absolute bytes>
        <10 skip> <code relative fixup> <absolute bytes>
        <11 skip> <data relative fixup> <absolute bytes>
        <<if <token(2)> calls for a byte, word, or quad,
                   then 1, 2, or 4 bytes will replace the 2-byte token.
          If Signature = Word values, code/data relative fixups are 2 bytes
          If Signature = Quad values, code/data relative fixups are 4 bytes
        AND upper 4 bits of token = (reference) FixupKey.
        >>
```

The .OBJ Cross Reference Area

Following the final 0-byte and fixup count byte, the remainder of a .OBJ file consists of a series of word pairs giving an index to a reference point in the code and a token to characterize the reference. During an XPL translation the Cross Reference Area is necessary to contain all the fixup information for the object code which is emitted byte for byte. After translation, the fixup format is built with the combined information from the two areas, and the cross reference information becomes redundant. It can be used to generate a cross reference listing, or it may be omitted.

The XPL Translator

Like the other translators, XPL is built from a slightly modified root module, it shares the file read, write, and format routines, and borrows its MAIN module from PTRAN. Starting with the series of MORE modules, its code departs from that of the other translators. Of course, it also contains PTRAN and OTRAN data structures built from PTRAN, OTRAN, and DEXPL translations. These are interpreted by the MATCH and WRAPUP engines, which call upon a set of primitive functions unique to XPL. These data structures define each version of XPL as a unique source language with a unique target machine, and therefore a unique XPL translator.

With two exceptions, the translation of an XPL .SRC file into a .OBJ file is almost completely like the source to binary translations of PTRAN, OTRAN, and DEXPL. These exceptions have to do with block structure and values (both address values and data values).

XPL Values

We have seen how XPL values are defined (in PTRAN, page 15) and referenced in DEXPL. This defines aspects of both their syntax and their semantics. Values can be forbidden to have a leading + or –, they can be required to have one, or it may be optional. They may have an optional second operand added or subtracted from the first. Values may be specified in terms of labels, constants, or strings. Labels may be defined as complex expressions, points in the code, or the base address of a block of data. Labels may also be forward referenced or even left undefined. When a reference to a value occurs before that value is known, it may be assumed to be a byte value. Sometimes this will later prove incorrect. When this happens, XPL invokes additional passes over the source with definitions for all locally defined values.

In an XPL translation, when one of the value primitives is called, a secondary match and wrapup is invoked by the primitives as a collection. Instead of producing a normal token that refers to alt #, length, and @source, the token emitted is:

```
                    <prim> <code> < Key:Token >
```

where, <prim> is the code of the actual primitive called and <code> is the canonical form returned by the subordinate match.

The actual and canonical* codes are listed below. “Key:Token” was defined on page 22.

```
        .#  0C2 - a = Byte or Ext *        symbol [addop symbol]
```

```
+#   0C4 - b = Byte or Ext *        addop symbol [addop symbol]
-#   0C6 - c = (C2,C4)              [addop] symbol [addop symbol]
~#   0C8 - d = (C2)         symbol [addop symbol]
.#@  0CA - e = Known Word  *        symbol [addop symbol]
+#@  0CC - f = Known Word  *        addop symbol [addop symbol]
-#@  0CE - g = (CA,CC)              [addop] symbol [addop symbol]
~#@  0D0 - h = (CA)                 symbol [addop symbol]
.##  0D2 - i = Known Quad  *        symbol [addop symbol]
+##  0D4 - j = Known Quad  *        addop symbol [addop symbol]
-##  0D6 - k = (D2,D4)              [addop] symbol [addop symbol]
~##  0D8 - l = (D2)                 symbol [addop symbol]
     0EC - v = (C2)         symbol (no 2nd operand allowed)
     0F4 - z = (C2)         label (undefined & no 2nd opnd)
```

The token object segment is allocated as follows:

```
0000 TOKENS        (64)    4 bytes / token, $0 thru $E (plus 4 bytes)
0040 PREP  (16)
```

The “prep” area is merely the leading few tokens that remain from matching “prep.” It contains any tokens left from matching the leading part of an executable statement that is then handed over to a DEXPL match. If all “prep” alternatives fail, the location 040 will be zero (as initialized by INIT.P).

When a value is written as an expression (from a simple constant with a unary “addop,” through the more complex forms allowed as “primitives,” to the form allowed as an “expr” in the label definition syntax), the elements of the expression are all entered into the symbol table as simple labels or ordinal values. Each value and expression is only entered once. A reference to the value –128 is via an expression which contains a reference to the ordinal value 128. When the expression is evaluated by EVAL.T (in the “wrapup” phase of the secondary match), a complete Key and value are computed. When a value cannot be determined, a Key = 0 is returned from EVAL.T (preventing the value from causing the match to fail).

EVAL.T produces Key = 0 unless it encounters an expression that evaluates to an absolute value, or a Code or Data Relative value with an absolute offset. The code contained in EVAL.T is duplicated in LINK to allow the final values of Symbol table expressions to be computed at LINK time.

XPL Block Structure

Every line of XPL has a “current indent point.” No statement body may begin in columns 1-8. As we have seen, columns 1-5 are reserved for XPL to report its results. Columns 6-8 are reserved for a statement label, so column 9 is the first legal indent point. When a program begins, the “current indent point” is initialized to zero. The first indent must be to column 9 or beyond (generally to column 17). Each subsequent line must indent 3 or more spaces beyond the previous line (eight spaces is typical), or use the same indent as some previous line. No indentation may exceed column 127 of a line of source. This provides for a maximum of 40 distinct points of indentation, which is several times more than is necessary (six indents being a typical maximum).

A statement has a reference prefix if its body is preceded by an * followed by at least two spaces. If it does not have a reference prefix, but requires one, its reference prefix is considered to be at the same column as its body. A statement that has a reference prefix, but does not require one, produces an error. If a line is indented to an illegal column, an “indent” error will be flagged. Reference prefixes and indentation points must match exactly. Each increase in indentation causes a block to be opened. Each decrease causes one or more blocks to be closed.

The DEXPL spec, “ref = mod” allows values derived from local block structure to be emitted into the code. This is handled by the N.MOD function. During pass 1, XPATCH entries bind an indentation level to three tokens: One each for the block’s LOOP, NEXT, and EXIT points. These tokens are defined by actions taken in the NEXT.LINE function.

XPL Translation

The main logic of an XPL translation is identical to that of PTRAN, OTRAN, and DEXPL as evidenced by the fact that they all share the same code for their MAIN module. Each has a different set of functions called from that main program, however. These include INIT.F, INIT.P, INIT.B, INIT.T, NEXT.LINE, and FINAL. Also included, of course, are the primitive functions referenced by both "match" and "wrapup." Here we will visit in a top-down sequence all of the routines that need to be understood before the final details of the XPL program code itself could be understood, modified, or copied into a derivative program.

Standard Initialization

Before INIT.F is called, certain processing is common to all of the translators in the XPL family. This includes initializing memory, opening a source file, displaying a usage or error message if this fails, and reading the source file into memory if the operation succeeds. The date-time of the source file is recorded so that it can be reset after the file is written back out, and so that the .OBJ file can be given the same date-time. This feature allows date-time to be associated with the latest edit changes to any source file of a program, and insures that is the date-time assigned to the final .COM file. Next, a routine called FIX.TEXT is called to put the source file into a standard form (expanding HTs into spaces, etc.). The stack pointer is not changed from the default given it by the OS. The source file begins with the next paragraph of memory above the 64K code (and stack) segment. Fixed data within the code segment are assigned at 0100 and overwrite the copyright string and the initial branch to MAIN (at the standard entry point of 0100). Approximately 160 bytes are available in this area.

The INIT.F function

This function does all the special initialization that occurs only once and isn't done by the standard functions. The copyright area is zeroed. The symbol table is initialized with the signature entry containing the "filename" of the source file. And, a list of index and address pointers are initialized.

The INIT.P function

This function is called once before matching each new line of source. It initializes "match" with the PREP data structure. It also initializes the source and token pointers. And it resets certain flags and values for processing a new statement. In addition, it must check a condition left over from the previous statement, namely whether the previous statement was a >prefix statement. In this case, another body must follow on the same line as the >prefix statement. This condition is signalled by the LSB of FLAGS.

A couple of observations are necessary at this point. The source pointer indicates the next byte of source in a 64K window. The window is advanced by NEXT.LINE. The tokens output from "match" are emitted into a 4K token segment, which is reused for each statement. The target of "wrapup" is two full 64K segments, one receives absolute object code, the other receives "tokenized" object code—byte, word, or quad values that are not evaluated when first produced.

The INIT.B function

This function initializes "match" with the first (next) body that comes from the DEXPL data structure. It knows that the source pointer is correctly set, but that certain flags, values, and the token pointer must be reset. It then checks if the first or next DEXPL body needs to be matched, or if there are no more alternatives it returns failure to the main processing loop. Normally, a function immediately followed by "match" needs to insure that the source and object pointers are correctly set, that DE = the address of the subpattern vector, and BP = the initial pattern to be matched. For XPL, however, things are more complex. If the first body definition is about to be matched, initialization involves more than it does if the next alternative is being sought. In both cases, the first and next alternatives are only found by following several pointers that are the leading part of the DEXPL data structure.

Above, the discussion of DEXPL did not define its object data structure. You may have gathered that it had components of both the PTRAN and the OTRAN data structures. If so, you were correct. The complete definition of the DEXPL data structure is given below (following INIT.T).

The INIT.T function

This function initializes “wrapup” to follow a successful “match.” First, it must determine whether the match came from a PREP (and needs to “wrapup” BACK), or came from a “body” that is followed in the DEXPL by a “back” unique to that statement definition. In either case, absolute object will be emitted to the OBJECT segment, and tokenized object will be emitted to XPATCH. Access to these segments is controlled by a set of functions contained in “wrapup.”

The Data Structure at DEXPLDEF

DEXPL is hard-coded to emit a .OBJ file that defines three Global symbols: PARMBYTE, BASEWORD, and DEXPLDEF. Parmbyte is equated to the signature byte defined by the DEXPL source. It determines the upper four bits of the signature byte given to every .OBJ file produced by a particular version of XPL. Baseword is defined as a hex constant in the DEXPL source. It determines the default listing base and the logical address of the first byte of code to be emitted (unless overridden by a CODE= or @PAD= declaration). At the address defined by the label, DEXPLDEF, is the following data structure:

```
DEXPLDEF       ENTRY
               =@vector0
               =@sub1
               =@sub2
               ...
vector0:       =t,c,@def1,@def2,..
vector1:       =b,c,@def3,@def4,..
               ...
sub1: =<ptran definition>
def1: =<ptran definition><flag><otran definition>
def2: =<ptran definition><flag><otran definition>
               ...
sub2: =<ptran definition>
               ...
```

Given the handle, @DEXPLDEF, the subpattern vector is located at handle+2, just as it is in a standard PTRAN data structure. However, instead of the root definition being the first address in the structure, “vector0” addresses the first set of statement beginners. This vector contains a list of the addresses of all statement definitions that begin with a non-keyword character. The second byte in each vector is a count of the addresses that follow it. All vectors after the first begin with a byte code equal to a keyword character that is one of the valid ways to begin each of the statements addressed in that vector. Given that a statement starts with a given character, it must be matched either by the list of definitions in vector0, or by the one that begins with that byte code. This means that only two vectors need to be considered for each statement match. Both vectors are attempted in turn, trying each alternative in the vector until a successful match is obtained, or until all the elements have been tried.

Because of the way they are coded, PTRAN definitions stop short of any PTRAN definition that immediately follows them. This would occur if the emit part of a DEXPL statement definition were empty. Otherwise, “flag” (which is a single byte) has its upper bit set to cause the PTRAN definition to stop short. OTRAN definitions define specific stop codes that do not depend upon the data structure that follows to make them stop short. In the DEXPL generated PTRAN definitions, no pattern begins with an override indicator, so the 040 bit is never set in the leading byte. This bit is used to distinguish a “flag” byte from a PTRAN definition. If the bit is clear, an empty OTRAN definition is assumed, and a pointer to a zero-byte is returned from INIT.T to cause an immediate return when “wrapup” is called. Otherwise, the bit is set and the byte is interpreted as a “flag” byte followed by a non-empty OTRAN definition. The least significant six bits of the flag byte are used to initialize the default flag that will be written into column 5 of the statement just matched (the possible “flag” characters were defined above). A pointer to the next byte is returned from INIT.T to “wrapup.”

DEXPL generates only a subset of the PTRAN and OTRAN pattern codes, so the definition of these codes as given above is more than complete. To summarize, the PTRAN codes not used by DEXPL are the option and repeat brackets, and the leading override (indicated in PTRAN with a hex number to the left of the = in a definition). The OTRAN codes not used include all those involved in token pointer movement and token relative conditionals.

DEXPL patterns assume the presence of a set of primitive functions in the XPL “match” and “wrapup” environments. For example, the symbol .# represents a byte value with no leading + or - operator. DEXPL replaces this symbol with a reference to the “a” primitive in the PTRAN code it generates. When an XPL “match” occurs, a call to the “a” primitive must match any such legal byte value in the XPL source. This, of course, is true for all the “match” and “wrapup” primitives. In brief, the primitives are defined as follows:

```
MATCH PRIMITIVES:
a - .#     Byte Ordinal    symbol [addop symbol]
b - +#     Byte Integer    addop symbol [addop symbol]
c - -#     Byte Int/Ord    [addop] symbol [addop symbol]
d - ~#     Byte SelfRel    symbol [addop symbol]
e - .#@ Word Ordinal       symbol [addop symbol]
f - +#@ Word Integer       addop symbol [addop symbol]
g - -#@ Word Int/Ord       [addop] symbol [addop symbol]
h - ~#@    Word SelfRel    symbol [addop symbol]
i - .## Quad Ordinal       symbol [addop symbol]
j - +## Quad Integer       addop symbol [addop symbol]
k - -## Quad Int/Ord       [addop] symbol [addop symbol]
l - ~##    Quad SelfRel    symbol [addop symbol]
m - match previous token (primitive')
n - extended function (1-4 byte parameters follow)
o - mandatory continuation
p - end-of-line
q - match a filename (for NAME= and LOAD=)
r - comment or ignored line
s - "succeed" if = in column 5 or 6
t - match 2+ spaces
u - set source pointer to column 6
v - match a value (literal or symbol)
w - match a long string (3+ characters)
x - set columns 1-5 to blanks
y - set pointer to <body> & columns 1-4 to hex of C.LIST
z - match a label

WRAPUP PRIMITIVES ({ | } ~ ? are undefined):
` - force an error during wrapup
a - $1 \
b - $2  \
c - $3   \
d - $4    |        Each of these primitives replaces an EMIT
e - $5    |        reference to a Byte, Word, or Quad value
f - $6    |        defined by matching a primitive (a-l),
g - $7    |        or by a definition in the "spec" clause.
h - $8    |
i - $9    |        They each refer to an absolute token. It is
j - $A    |        not possible for $0 to refer to a value, so
k - $B    |        $0 is not defined.
l - $C   /
m - $D  /
n - $E /
o - a state-change function that enforces statement order rules
p - handle any @PAD=<hex> or .PAD=<hex> statement.
q - handle NAME=<filename> (rewrite signature entry in SymTab)
r - handle the [<value>*]<value> in a code emit
s - handle a <long string> in a code emit
t - handle additional (LOAD=) <filenames>
u - extended function (precede with EMIT code)
v - handle <value> primary in an <expr>
w - handle <expr> (subexpression) in an <expr>
x - handle any CODE= | DATA=
```

XPL MATCH

Patterns generated by DEXPL automatically use the primitive functions “a” through “p” to represent the twelve forms of XPL “values” and the other things that DEXPL allows you to define, such as the rematch of a given token, mandatory continuation (for a multi-line statement definition), end-of-line at the end of every definition, and the general purpose "n" primitive that represents all "spec" and other modifier operations.

The "n" Primitive

The "n" primitive is used to represent all of the special actions that DEXPL can represent that occur during a pattern match. "n" is responsible for interpreting the codes that follow it in the pattern structure, and for advancing the pattern pointer for "match." "n" is followed by from one to four bytes that are encoded as follows:

```
[0000 0000]                    = >prefix statement
[0000 0001]                    = NEXT attribute
[0000 0010]                    = GLOBAL attribute
[0000 0011]                    = FPFX flag
[0001 $ref] [mod/mod/mod]  = $ref=mod[,mod][,mod]
 where, $ref =     0000..1110 for $0.. $E
        & mod:     0001xxxx = BYTE 00xx01xx = ORD  00xxxx01 = LOOP
                   0010xxxx = WORD 00xx10xx         = INT   00xxxx10 =
NEXT
                   0011xxxx = QUAD 00xx11xx         = REL   00xxxx11 =
EXIT
[0010 $ref] [byte][flag]         = ($ref=b flg)
[0011 $ref] [byte][byte][flag]     = ($ref=b,b flg)
[0101 $ref] [byte]          = <$ref=b>
[0110 $ref] [byte][byte]         = <$ref=b,b>
[1001 $ref] [byte]          = <& $ref=b>
[1010 $ref] [byte][byte]         = <& $ref=b,b>
[1110 $ref] [byte(1)][byte]  = $ref=(1)b
 where, (1) is the field length in bits of the binary value, b.
           All byte sequences are the same as in their syntax.
```

XPL WRAPUP

After a PREP and/or BODY match, a sequence of tokens and other data structures defines the current statement being translated and the complete context for its translation. WRAPUP drives this process from the OTRAN code that corresponds to a statement defined by DEXPL, or from the PREP data structure defined by PTRAN. In the former case, tokens are in fixed positions, and the OTRAN data structure contains constants to be emitted, fields that are alternative numbers to be shifted and XORed into the last emitted constant, or multi-byte values that are emitted as a result of one of the "wrapup" functions "a" through "n." The other "wrapup" functions are referenced in the BACK data structure that corresponds to PREP. These functions do very specific things related to meta-statements and declarations.

One thing that requires further explanation, however, is the implicit ordering between the absolute token sequence that is generated by a particular DEXPL statement definition and the EMIT definition that corresponds to it. In the EMIT definition, there may be $ref symbols used in two entirely different ways. Inside [bracket] EMIT definitions, a $ref indicates a bit field derived from matching a statement or pattern alternative. Outside, a $ref is a byte, word, or quad EMIT derived from matching a "value" primitive. There must be no ambiguity between which $ref indicates one or the other of these, nor which in a sequence of several.

To insure that this is the case, a canonical sequence is defined, and each $ref is paired with a "value" or with an alternative that represents a bit field fixed (1 to 8 bits) in size. Many of the possible mistakes in correctly pairing a particular $ref with the appropriate token are caught by the DEXPL translation. In other cases, DEXPL must take specific actions to insure that a canonical pairing exists. Consider the following:

```
                   $index=reg
where,
              index   = HL | BP | SI | DI
              reg     = A′ | A | B | C | D | E | H | L
```

Here, three $ref tokens are defined. $0 refers to the 1 alternative statement definition itself. $1 refers to the alternative of "index" that matches, and $2 refers to the alternative of "reg" that matches. Moreover, $0 refers to a (rather useless) 1-bit field, $1 refers to a 2-bit field, and $2 refers to a 3-bit field. These are given by the number of alternatives possible for each of them. Now consider the above statement definition with a different definition for the "index" pattern:

```
              index   = (HL+#) | (DI+#) | SI | DI
```

This changes everything. Now, if $1 = 0 or 1 (one of the first two alternatives of "index" matches), then a byte "value" is defined and should be referenced by $2. If one of the last two alternatives of "index" matches, no "value" is defined, and $2 continues to represent the alternative of "reg" that is matched. Here, DEXPL must step in to define a canonical ordering, so that all possibilities are considered and each $ref corresponds to a unique token. In this case, it recognizes the "+#" primitives in the definition of "index" and reserves $2 uniquely for them. This means $3 will represent "reg" no matter which alternative of "index" is matched. Now, let's add one further complication. Let's say that the statement definition had two alternatives, as follows:

```
$index=reg
reg=$index
```

Now, the problem is that the second alternative reverses the order of "index" and "reg." This defines a different pairing with $1 through $3. Again, DEXPL steps in to give all subsequent alternatives the same pairing as the first. Therefore, $3 refers to the alternative matched for "reg" in both alternatives of this definition. It need not be the case that the same subpattern or primitive is referenced in each alternative of a higher level definition, only that bit fields of equal sizes be paired in the canonical ordering, and that "values" be paired with "values." When an EMIT contains a $ref for a "value," a byte, word, or quad EMIT may result, depending on what actually matched. When a $ref refers to a bit field, it must refer unambiguously to a field of a fixed size. DEXPL insures that all the fields of a byte EMIT add up to exactly eight bits.

Other Modules & Functions

The NEXT.LINE function

This function initializes the source pointer for the next statement match, or returns CY if no more lines of source remain. If FLAGS indicates a >prefix statement, the segment pointer, DS, is not advanced. Otherwise, SI is reduced to a minimum, and DS is increased to compensate. The BODY.PTR and COLUMN.1 indices are set equal to FILE.PTR. If fewer than 9 bytes remain in the source file, an end-of-file condition is returned via the CY flag. As explained later, NEXT.LINE in XPL also handles block structure and any subsequent passes over the source.

The FINAL routine

This routine builds the final .OBJ data structure. If errors were detected, it does nothing. If a .OBJ file is to be produced, it must update the symbol table, and merge OBJECT and XPATCH into a "fixup" data structure. The MAIN routine FINAL.OBJ will scan the symbol table and fixup area and insert the count bytes at the appropriate points. This process insures that a proper .OBJ file is delimited. Other MAIN routines determine the .OBJ filename and write the file (giving it the same date-time as the .SRC file had before translation).

The Base Template

This is the module named PTRAN, OTRAN, DEXPL, or XPL, as the case might be. It serves as the LOAD root for the other modules of the respective programs. It contains the copyright string, some primitive display functions, and a branch to the MAIN entry point. It also names the next tier of modules to be linked (LF, FF, SF, and MAIN). Except for minor differences in the Base Template, these modules are identical over all of the four programs.

LF - Load File

This module contains the code that accesses and reads a source file in a standard fashion. It is used unchanged by all the XPL family of programs.

FF - Fix File

This is a text file fixup program that standardizes spacing and end of line characters. It converts tabs to spaces before translation, and spaces back to tabs after translation is complete.

SF - Save File

This standard module writes files. The source and .OBJ files are given the same date-time as the source file has when originally read by LF.

MAIN

This standard module was explained in some detail on page 6. It contains a LOAD= to the MORE routine.

MORE

This is the first non-standard module in each translator. It contains a LOAD= to PREP, BODY, BACK, XPLDEF, and USAGE. Of course, as explained above, PREP, BODY, BACK, and DEXPLDEF are data structures generated by the PTRAN, OTRAN, and DEXPL translators. MORE may also contain a LOAD= to MORE2, MORE3, and so on. In general, the “match” primitives are contained in MORE. MORE2 and higher modules contain functions for “wrapup” and “final.”

USAGE

This module contains all display strings and other code that seldom, if ever, gets accessed in debug. It is the last module linked and defines @End-of-Code for initializing the heap.

Symbols and Addresses

Throughout an XPL program, labels may be defined and referenced. A label may be defined once within a source module, or not at all. If a label is undefined with respect to a given source module, it must be defined as a GLOBAL in exactly one other source module within the collection of modules that are to be linked together into a program. Each label may be defined in one of three ways: It may label an expression, a data declaration, or a code declaration. An expression allows a label to be defined as an absolute value, or in terms of other labels. A data declaration accomplishes two results: It assigns the next data address to the label, and it reserves a given number of bytes above that address before the next data address assignment. In all other cases where a label appears in the definition position of a line, it is defined with the next code address.

Each of the three types of definition has two different contexts: One assigns the LOCAL attribute to the label, the other assigns the GLOBAL attribute. For both expressions and data declarations, all definitions made while STATE = 4 declare labels with the GLOBAL attribute. All with STATE = 5 declare labels with the LOCAL attribute. There is no way to override this action for declaration statements. Labels attached to DEXPL defined statements get the attribute LOCAL by default, or GLOBAL if the DEXPL definition contains that keyword. The effect of the GLOBAL attribute is that the label becomes visible to other source modules. LOCAL labels, on the other hand, may be defined in several source modules without any confusion. There are only these two levels of scope in XPL.

In addition to the LOCAL and GLOBAL attributes, labels may have a variety of other attributes. In addition to the attributes that a label may have, references to labels may have even further attributes. Attributes (like LOCAL and GLOBAL) may be mutually exclusive; other attributes may be combined. Another set of four attributes that are mutually exclusive are: absolute, code relative, data relative, and undefined. When a label is undefined at the end of a source module, its attribute becomes EXTERNAL (it must be defined by a GLOBAL in some other module). All labels must eventually be defined as absolute, code relative, or data relative values. The job of LINK is to define both code and data relative values as absolute values. Another set of attributes is Byte, Word, and Quad. Still another is Integer, Ordinal, and “self-relative.” These aren’t mutually exclusive; they overlap. A Byte value is also a Word or Quad value, and a Word value is also a Quad value. Likewise, positive integers overlap with ordinal values, and self-relative values are simply integer values computed in a special way.

When a label is referenced, the context of reference may impose certain constraints on the label’s definition. All forward references that demand a byte value are “conditional bytes” until the end of the source module is reached. At that point, they are considered external. When labels are defined, they must accord with all conditional bytes, otherwise, a Major Backup will be performed and retry will find the label defined as a word or quad.

Absolute Addresses

When a program is executed, all values are absolute. Absolute values (Byte, Word, or Quad) may be emitted by the XPL translator, or may be substituted by LINK. Certain conventions (beyond the scope of XPL) often exist to permit the operating system to substitute absolute values when it loads and executes a program.

Physical Addresses

There are three types of addresses in an XPL translation: Physical, logical, and listing addresses. A physical address is where a particular value is actually stored in memory or in a file. This may be during translation, in the .OBJ file, in the .COM file, or during the actual execution of the final program. A logical address refers to the final physical address. A listing address hopefully corresponds to it, too, but it may not. XPL allows complete control over both the final physical addresses and the listing addresses. Intermediate physical addresses are of no concern to the programmer, but they might concern the XPL developer. During translation, only absolute code is emitted into the OBJECT segment. An XPATCH area is used to record points where logical values are to be inserted. In the .OBJ file, these two streams are merged. The .COM file contains an absolute block of code with absolute addresses. There is (by convention) a fixed load offset between physical bytes of code and their final position in memory when the code is executed. Absolute address assignments anticipate the final position of code in memory.

Listing Addresses

The first four characters on each code generating or label declaring line of XPL are replaced as follows: The least significant word in the current offset of code or data relative address assignment is written according to whether the line is a code or data declaration. If the line defines a label in terms of an expression, these characters are blanked. Notice that the "listing address" is actually made a physical part of the source file itself.

Data Addresses

The "current" data address at the beginning of a source module is data relative with an offset of zero. Data declarations assign the current data address to a label, and increase the current data address by zero or more bytes. The DATA= declaration can reposition the current data address to any address that can be resolved to an absolute address by LINK. During translation, a data label is defined as the sum of the Key:Token value currently contained in the data word D.BASE(2) and the value contained in D.LIST(4). A reference to the label causes the Key:Token of this sum to be emitted into the code. When a data declaration reserves space, the value of D.LIST is increased by the amount reserved. The Token field of D.BASE is zero when absolute, code, or data relative values are in effect. The Token is non-zero when the <value> of a DATA=<value> or <label>(<value>) cannot be computed at the time of the XPL translation. The value must be computable at LINK time, however. When Token>0, Key=0, and vice versa.

Another value is contained in D.USED(4). This is the maximum value obtained during translation by D.LIST when D.BASE > 0C00 (data relative addressing is in effect). This value is passed to LINK to tell it how much space to reserve for the entire program module. LINK pads data address assignment between modules so that each module begins at a new memory paragraph (an even 16 byte boundary).

Code Addresses

Code address assignment is similar to the above, but it involves one addition: The @PAD value. This defines the initial listing address for code in the module. The default @PAD is given by the BASEWORD parameter defined in DEXPL. An override value may be given by the XPL programmer using the @PAD= declaration (in which case it is also passed to LINK). @PAD also affects the upper limit of .PAD= declarations and the way the module is handled by LINK. All of these can also be affected by a CODE= declaration.

The current code address is defined by C.BASE(2) + C.LIST(4). These are interpreted exactly like their Data counterparts. Two other data locations are advanced independently of C.LIST as code is emitted: XPATCH and OBJECT. Each time a Token value (byte, word, or quad) is emitted, XPATCH is advanced by two words. These contain $:Delta and Key:Token (explained later, under "Details"). Each time an absolute byte of code is emitted, OBJECT is advanced one byte. C.LIST is advanced one for one with OBJECT, and by the amount indicated by the Key fields in XPATCH. The update of C.LIST is performed by NEXT.LINE.

When a .PAD= or CODE= is encountered, a special Token is emitted to XPATCH. This is called an Org Code. It records for LINK a new address to which the next byte of code is to be located in the .COM file. Physical code continues at OBJECT. The values in C.BASE and C.LIST are reset with new values determined by the .PAD= or CODE= declaration and a special Key:Token is emitted into XPATCH (Key = 0, 4, 8, or C).

Now let's take a close look at what the PAD and CODE declarations do. First, why do we need them? The CODE= declaration allows programmer control over where the code is located in the final program. During debug, it is essential to have a correspondence between your program listing and the actual code in the machine. As you update a program, it shifts around in memory. Unless you have a huge screen or a dual screen computer, you will probably use a hardcopy listing of your program while you work in the computer's debugger. It's nice if every little change to your program doesn't require a new listing of the entire program. This can only happen if some kind of "padding" is adjusted between pages and between modules. XPL allows you to specify padding so that the listing stays current on all but the pages that have actually changed. When the padding is used up, a change may ripple across several pages. When one module bumps into the next, the modules may ripple. But, even when ripples occur, padding keeps the least significant part of the listing addresses consistent with the final code.

So, what's involved? First, is the default action. If an @PAD= declaration is omitted, C.LIST is initialized to BASEWORD and C.MASK is set to zero. No C.MASK is passed to LINK, nor does an Org Code begin the program module. LINK assumes the BASEWORD default as the first logical address of the module, and it adds no padding in front of the module. Each module's Code Relative address space is independent of all others, so LINK merely subtracts any explicit or implicit initial offset from all Code Relative addresses and then adds back the absolute address it has assigned to the base of a module to get its final absolute addresses.

If an @PAD= is present, it occurs very near the beginning of a program. The @PAD declaration defines the initial values of C.LIST(4) and C.MASK(2). An @PAD=0 sets both of these to zero. An @PAD always causes an Org

Code to be emitted. Any C.MASK > 0 is passed to LINK. The default of LINK is to locate the root module of code at the address given by BASEWORD, and to pack subsequent modules with no padding between them.

The C.MASK derived from an @PAD= declaration contains 1 bits in each position where the @PAD= has contiguous 0 bits at its right. For example, @PAD=0300 has 8 zero bits at its right. This would set C.MASK to 0FF and C.LIST would start at 0300. LINK uses C.MASK to pad in front of any module that declares it. If it is declared correctly, this will produce the address assigned to C.LIST and keep the listing equal to the address assignments during a sequence of code updates. The mask defined by the pad (1 bits for contiguous 0 bits at the right part of the hex constant), is ORed into the current address and one is then added, so that the next address assignment has the same number of 0 bits as the pad. The .PAD= declarations allow padding between pages of code within the current module. Here, only the least significant zero bits have any effect, and those only if they are less than or equal to the number declared in the original @PAD=. If no @PAD= is declared, or if a CODE= has declared an absolute or data relative base, a .PAD will not have the same effect. In the absence of an @PAD= if code relative addressing is in effect (or whenever an external base is in effect), a .PAD will cause no pad at all. If absolute addressing is in effect, a .PAD may be as large as 08000. If data relative addressing is in effect, all .PADs will pad only to the next paragraph.

Here are the data declarations used to manage address assignment, with a comment on each:

```
0000 D.USED       (4)     Saves max DRA assignment over a translation
0000 C.MASK       (2)     Mask value from @PAD= declaration (or 0)
0000 C.BASE       (2)     Key:Token of Code Address Base
0000 C.LIST       (4)     Current Offset of Code Address assignment
0000 D.BASE       (2)     Key:Token of Data Address Base
0000 D.LIST       (4)     Current Offset of Data Address assignment
```

The values stored in C.BASE (Key:Token) and D.BASE, (Key:Token), are the same as those used in a reference context (as defined on page 22). Values with the Token field = 0, are used in C.BASE and D.BASE to indicate that absolute, code relative, or data relative addressing is currently in effect (Token > 0 means that an external base is currently in effect). Key (in C.BASE and D.BASE) is never 1, 4-9, 0C, or 0D. It may only be 1-3, 0A, 0B, 0E, or 0F.

Caveat Emptor

The ability to define the current code and data addresses, in terms of general expressions that may involve forward references and externals, allows the ability to express circular references and meaningless operations. In general, it will be up to LINK to make final address computations and report any symbols it can't define. Some meaningless operations are detected and reported by XPL, some are passed to LINK, and the rest will result as bugs in the object program.

The Complete XPL “prep” Definition (and ASCII codes)

```
0000 prep  = .PAD= pad p |
0000                r |
0000                NAME= q [t files] p |
0000                LOAD= t files p |
0000                @PAD= pad p |
0000                CODE= expr p |
0000                DATA= expr p |
0000                s |
0000                u z [t] x def p |
0000                u [z] [:] [t] y = string [, string].. p |
0000                u z [t hd xd xd hd z].. [t] p
0064 pad          = hd [hd] [hd] [hd] [hd]
0072 hd           = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
0072                8 | 9 | A | B | C | D | E | F
0092 def   = ( v ) | "=" expr
0099 files = q [, q]..
009F string       = [v *] value | "'" hd hd [hd hd].. "'" | [fix] w
00B3 value = c | g | k
00B9 expr  = [unary] primary [binary primary]
00C2 unary = + | - | ~
00C8 binary       = + | - | * | / | \ |= MOD | `* | `/ | `\ |= `MOD |
00C8                      & |= AND | "|" |= OR | ! |= XOR
00F5 primary      = v | ( expr )
00FB fix   = | ` | ^ | `^ |= ^` | ~
010B xd           = hd | T | O
<<
MATCH PRIMITIVES:
          a-l = .# thru ~##
          m   = match prim' (string at saved token position)
          n   = extended function (with 1-4 parameter bytes)
          o   = mandatory continuation
          p   = End-of-Line
          q   = filename
          r   = comment or ignored line (if = in col 5/6 then fix)
          s   = succeed if = in col 5/6.
          t   = match 2+ spaces
          u   = set source pointer to column 6
          v   = match a value (literal or symbol)
          w   = match a long string (3+ characters)
          x   = set columns 1-5 to blanks
          y   = set pointer to <body> & columns 1-4 to hex of C.LIST
          z   = match a label

-----------------------  ASCII CODES  -----------------------
00 ^@      08 ^H   10 ^P   18 ^X   20 SP   28 (    30 0     38 8
01 ^A      09 ^I   11 ^Q   19 ^Y   21 !    29 )    31 1     39 9
02 ^B      0A ^J   12 ^R   1A ^Z   22 "    2A *    32 2     3A :
03 ^C      0B ^K   13 ^S   1B ESC  23 #    2B +    33 3     3B ;
04 ^D      0C ^L   14 ^T   1C ^\   24 $    2C ,    34 4     3C <
05 ^E      0D ^M   15 ^U   1D ^]   25 %    2D -    35 5     3D =
06 ^F      0E ^N   16 ^V   1E ^^   26 &    2E .    36 6     3E >
07 ^G      0F ^O   17 ^W   1F ^_   27 '    2F /    37 7     3F ?

40 @       41-5A = (A-Z)   5B [    5C \    5D ]    5E ^     5F _
60 `       61-7A = (a-z)   7B {    7C |    7D }    7E ~     7F ^BS
      ^ means hold CTRL down while pressing indicated key.  >>
```

The Complete XPL “back” Definition

```
0000 back:  $=0  o EMIT(44,55,EF) p;
0008           $=1  ;
000A           $=2  o EMIT(01,23) q + files;
0014           $=3  o EMIT(01,11,23,33) files;
001E           $=4  o EMIT(02,13) p;
0025           $<7  o EMIT(55,CD,DD,E4) expr x;
0030           $=8  o EMIT(55,CD,DD,E4) + + def;
003C           $=9  o EMIT(CD,DD,E5) bump + + emit;
0049           $=A  o EMIT(CC,DD,EC);
0050                `.
0052 files:              + t + rep.t.
0058 rep.t:                t + :rep.t
005C def:  $=1     expr.
0060 expr:       + EMIT(0,0,0,0,0) unary primary binary.
006E unary:                XOR($,4).
0070 binary:               XOR($) XOR(80) primary.
0076 primary:      $=0     v +;
007A                     + expr w.
007F emit:         string rep.string.
0084 string:       $=0     bump + r +;
008B               $=1     EMIT() XOR($,4) XOR($) hex;
0092                     + bump s +.
0098 rep.string:           string :rep.string
009C bump:       + .
009E hex:          EMIT() XOR($,4) XOR($) :hex
<<
WRAPUP FUNCTIONS:
  ` - Prep Error ($=7 forces Prep Error at WRAPUP)
  a = $1   Used to implement the $ref functions to
  b = $2   EMIT a # (Byte), #@ (Word), or ## (Quad).
      ...
  m = $D   Special $ref for use as trap during Debug
  n = $E
  o = a state-change function that enforces statement order rules
  p = handle any @PAD=<hex> or .PAD=<hex> statement.
  q = handle NAME=<filename> (rewrite signature entry in SymTab)
  r = handle the [<value>*]<value> in a code emit
  s = handle a <long string> in a code emit
  t = handle additional (LOAD=) <filenames>
  u = extended function (precede with EMIT code)
  v = handle <value> primary in an <expr>
  w = handle <expr> (subexpression) in an <expr>
  x = handle any CODE= | DATA=
STATE:
           0 = Begin Program
           1 = Have NAME= and/or LOAD=, may not get another NAME=
           2 = Have @PAD=, may not get another @PAD=
           3 = Have NAME= and/or LOAD= and @PAD=, may ONLY get LOAD=
           4 = Have GLOBAL, no NAME=, LOAD=, or @PAD= allowed.
           5 = LOCAL context, no NAME=, LOAD=, or @PAD= allowed.
           .PAD= only allowed in STATE=4,5 (0EF forces "ignored" line)
           EC = flag begin XREF; CD = Seq error; DD = Seq error.
>>
```

The Complete XPL "primitives" Definition

```
0000 match.prim = c4 | c2
0004 c4     'C4'= unary symbol [binop symbol]
000C c2     'C2'= symbol [n'0A' binop symbol]
0015 symbol  = [pfx] z | binary | "'" hd hd [hd hd].. "'" |
0015              [fix] [pfx] decimal | [fix] [pfx] hex | [fix] [pfx] w
003D unary = | + | -
0042 binop = + | -
0046 binary = bd bd bd bd bd bd bd bd [bd bd bd bd bd bd bd bd] n'0B'
005B bd    = 0 | 1
005F decimal  = cd [dd] [dd] [dd] [dd] [dd] [dd] [dd] [dd] [dd] n'0B'
007E cd      = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
0091 dd    = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
00A5 hex   = 0 [hd] [hd] [hd] [hd] [hd] [hd] [hd] [hd] |
00A5                 cd [hd] [hd] [hd] [hd] [hd] [hd] [hd]
00D6 hd    = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
00D6                 8 | 9 | A | B | C | D | E | F
00F6 pfx   =|| @ | #
00FC fix   = | ` | ^ | `^ |= ^` | ~

0000 wrap.prim:   EMIT()
0001                 $=0   XOR($) EMIT() pfx sym EMIT(E) opt u;
000E                 $=1   XOR($) EMIT() sym EMIT(E) opt u.
0019 opt:          - EMIT(C4,0) fix sym EMIT(0C).
0024 sym:    EMIT()
0025                 $=0   pfx EMIT() u+;
002C                 $=1   EMIT(2) u;
0031                 $=2   EMIT(4) u;
0036                 $=3   fix pfx EMIT(6) u;
003F                 $=4   fix pfx EMIT(8) u;
0048                 $=5   fix pfx EMIT(A) u+.
0052 pfx:          XOR($,4).
0054 fix:          XOR($).
<<
Special Match Primitives for handling DEXPL Primitive Symbols:
     n'0A' = succeed if P.V entered via P.A thru P.L, else fail.
     n'0B' = fail if next char is any "hd" else succeed.

Special Wrapup Functions for handling DEXPL Primitive Symbols:
     00'u' = <label>
     02'u' = binary
     04'u' = decimal
     06'u' = hex
     08'u' = 'hex'
     0A'u' = "string"
     0C'u' = bin +/-
     0E'u' = (no bin)
>>
```

The XPL Symbol Table

Each symbol table entry consists of a type:skip byte, a fixed field generally containing a 2-byte primary token, and a variable length byte string. The byte string of an expression or equate entry contains a secondary token or tokens (always identical to the primary token of some other entry). When a constant is referenced it is immediately recorded as a Byte, Word, or Quad Ordinal (all values are stored in Little Endian sequence, and are byte reversed when substituted by LINK if PARMBYTE indicates a Big Endian target).

Labels and expressions are referenced before definition occurs. The first symbol table entry binds a Token to a label or value. Later, a label may be bound to a value or expression by having its primary token duplicated as the primary of a second symbol table entry (either directly as the primary for the value or expression, or indirectly using an equate entry).

Symbol table lookup involves initializing a buffer with the byte string to be looked up, the length of the byte string, an entry group type, and an optional token. If the string is found, its primary Key:Token is returned. If not, an entry is made according to certain rules involving the lookup parameters.

There are two lookup groups: Labels (types 7x-Bx, creating a 7x if not found), and Values (types Cx and Dx). Lookup insures that when a label (value) is being looked up, a value (label) is not matched and returned by accident. This is done based on entry types. In neither case of lookup should a type 0x–3x, or Ex–Fx be looked up. The first group are not currently used during translation. A type E5 symbol is entered as an equate of one token to another.

Key:Token Codes

When a primitive is referenced in an executable statement, one of twelve permutations of the required value may be specified (these result from three forms of syntax, three sizes of value, and the ability to specify self-relative values). When a symbol is defined, one of ten specifiers is entered into the symbol table to define its value. These specifiers are called the Key of a Key:Token combination. The Key of the reference must accord with the Key of the definition. For example, if the reference primitive specifies a Byte Self-relative, the definition must be within +127 and –128 bytes of the code emit address (after the value is emitted). Self-relative values are entered as special expressions in the symbol table. When a reference Key indicates a Byte Integer and its corresponding definition is a Byte Ordinal with a value of 128, the actual value of the reference is –128. The Word Ordinal, 2000, could be referred to by a Key specifying any of the following: A Word or Quad Code (or Data) Relative value, or a Word or Quad Ordinal or Integer.

A Key:Token is a 16-bit value consisting of a 4-bit Key, and a 12 bit Token. In an evaluation context, Key = 0 indicates that the token's definition is unknown; Key = Token = 0 indicates an error. In the symbol table, Key > 0 is used in the primary positions of all entries where the definition of that token is known.

Constants

Constants are entered into the symbol table as one-, two-, or four-byte ordinal values. These are types D4, D5, and D7 entries. These entries may be referenced (that is, their tokens used) in code emits, as the primary token of a type Ax–Bx entry, or as a secondary token of an E5 (Equate), D6 (unary expression), or D8 (binary expression). As a primary token in a type Ax–Bx entry, or from a code emit, a token is always decorated with a Key—its usage is completely specified. All value definitions are considered to be Quad Ordinals, however an expression may result in any of the twelve basic types (Ordinal, Integer, Code, or Data Relative, across the three sizes, Byte, Word, or Quad).

Constants may be referenced explicitly as literals, or implicitly, in certain contexts, as Self-, Code, or Data Relative values. In all cases, an Ordinal value is recorded in a Dx entry. A reference in a code emit is recorded as a fully qualified Key:Token, whose Token part is the same as the primary token of the Dx entry. It is possible in code emits to use different Keys with the same Token. A Code Relative value = 1000, a Data Relative value, and a Word or Quad Ordinal or Integer could all have the same Token, even though they would all have different Keys.

Labels

A label entry is created when a given alphanumeric label is matched for the first time by the '00'u Wrapup Function (a type 7x symbol table entry is created), or is defined by the action of NEXT.LINE (type 8x–Bx). A new symbol table entry allocates a new token value contained in no other symbol table entry (at that moment). Different labels (and therefore different tokens) may be defined with the same value. There are four ways this may be done. Each causes an "equate" relationship between the tokens. The first method is a direct equate. The second and third ways are by writing two successive code or data declarations, the first of which allocates no new space. The final way is an indirect equate. Examples of each of these are as follows:

```
       FIRST=SECOND                   FIRST is (explicitly) equated to SECOND
```

```
LABEL1      (0)
LABEL2      (2)             LABEL2 is equated to LABEL1

LABEL3      NEXT
LABEL4      CALL SOMEBODY   LABEL4 is equated to LABEL3

THIS=1000
THAT=1000           THAT is (effectively) equated to THIS
```

Whatever definition is given to SECOND, that will be the definition given to FIRST. FIRST may not be otherwise defined. LABEL1, however, may be given a different definition (in place of 0) and the equate will be broken. Likewise, if a non-null code definition replaces NEXT in the definition of LABEL3, an equate will no longer exist between LABEL3 and LABEL4. The equate between THIS and THAT is the easiest to break. Either of the constants (1000) may be changed. Of course, none of these can happen within the same translation, they must be done by a later translation after some change is made to the source.

Each time a label is referenced before it is defined, its token is recorded somewhere—either in a code emit, or in another symbol table entry. In another symbol table entry it is recorded as 0:Token (Key = 0). However, in a code emit, it must be qualified as Byte, Word, or Quad, and as Ordinal, Integer, Code relative, or Data relative. Self-relative values are emitted as Key = Integer tokens that reference a special expression.

A label is defined when it begins in column 6-7. Its definition depends on what type of statement it labels. There are three possibilities: Data, Equate, and Code. Examples of the syntax are as follows:

```
LABEL       (5)     Defines LABEL as the next Data Address assigned
LABEL       = 5     Defines LABEL as the absolute value 5
LABEL       NEXT    Defines LABEL as the next Code Address assigned
```

All Data and Equate definitions follow a form similar to the first two examples, above. Code addresses are defined, in general, by a label on any other type of statement.

Changes to a symbol's definition can occur within a translation. For example, when a reference to ABC is first made, the symbol is entered into the symbol table (as a type 7x entry, via the n'0A' primitive). Later it could be equated to an expression that also contained an external or forward reference. Finally, all of the symbols in the expression might be defined. This would complete the evolution of the original symbol. The sequence might be as follows:

```
X=ABC               ABC is defined as unknown (a type 7x entry).
ABC=NEW+40 ABC is now a LOCAL or GLOBAL expression (8x or 9x)
NEW=-10             Now ABC is defined as +30 (an Ax or Bx entry).
```

When labels are defined in terms of expressions that cannot be evaluated to an Absolute, Code, or Data Relative value, the definition of the label is recorded as 0:Token (Key = 0, meaning "undefined") in symbol table entries of type 8x or 9x. At the end of pass 1, if a symbol does become defined, all references to it also become defined, and its entry type is updated to Ax–Bx, with fully qualified Keys.

Expressions

NEW, above, is defined in three steps. First, the constant 10 is defined as a byte ordinal. Next, a unary minus expression refers to that constant. When the expression is evaluated, the result is the byte integer, -10. Expressions are referenced from three contexts: As a primitive in an executable statement, as the body of a CODE=, DATA=, or <label>= declaration, or as a subexpression of an expression. The evaluation of an expression must be consistent with the context of its reference. For example, a primitive may specify Byte, Word, or Quad and the expression may not evaluate to Word for a Byte reference, nor Quad for a Word reference. Different value ranges apply to Integers and Ordinals. Many expressions cannot be evaluated until LINK has assigned absolute values to all symbols. For example, if ABC is Code Relative, then -ABC is undefined until LINK time. On the other hand, if both ABC and DEF are Code Relative, the value ABC-DEF is absolute, even though neither of its operands is. The value could also be Byte, even if both operands are Word or Quad.

The operands and operators of an expression define its value type. Expressions begin as Quad Ordinals, but a key is developed to type them further. For example, when a negative result is obtained, Ordinal is changed to Integer. When a positive Code or Data relative value is produced, Integer and Ordinal are changed to Code or Data Relative. The difference of two Code or Data Relative values is changed back to Integer. However, any value that cannot be computed is changed to Key = 0 (external). This would hold true for the negative of a Code Relative value, or the sum of two Code Relative values (and other similar expressions), as well as any expression involving an undefined symbol or subexpression.

Expressions and constants are only entered into the symbol table for purposes of reference. The most common form of reference is when a primitive is matched in a code emitting context. This limits the complexity of the expression. The other possibility arises from one of the statements: CODE=, DATA=, or <label> =. Each of these may employ nested expressions of varying complexity. As an expression is matched, it is entered into the symbol table. An attempt to evaluate it is made whenever a reference to it takes place. This produces a Key. A Key = 0 means the evaluation failed (the result is unknown). Otherwise, a Key = 1, 2, 3, 5, 6, 7, A, B, E, F is produced.

The twelve primitives (.# through ~##) allow three sizes of values (Byte, Word, and Quad), and three types of values (Ordinal, Integer, and Self-Relative) to be specified. Further, Integers may be required to have a leading plus or minus sign, or it may be optional. These options give us the twelve symbols.

The “spec” syntax in DEXPL also allows values to be defined or verified. When any of the keywords, LOOP, NEXT, or EXIT is used, a $ref=<keyword list> defines a value that can be Byte, Word, or Quad, and furthermore as Integer or Self-Relative (with Byte and Ordinal being the defaults). If the LOOP, NEXT, or EXIT keyword is missing, ORD, INT, REL, and BYTE, WORD, or QUAD are used to verify the value already present in $ref.

Self-relative values are always emitted as Integer Tokens of the appropriate size (Byte, Word, or Quad) that reference a type D6 Symbol table entry. The special OpKey = 070 is given to this type of expression. It always indicates the difference between the target address and the next code address.

XPL Data Areas

A 154-byte static data area is declared to overlay the code that begins at 0100 (just above the PSP). A jump to MAIN is coded at 0100, and the copyright string is coded above that. The jump is taken and the copyright string displayed when XPL first begins execution. After this, the data area is initialized.

The data area is accessed via direct addressing into the Code Segment from much of the code, but via BP relative addressing (into the Stack Segment, which is kept identical to the Code Segment) during the WRAPUP phase of statement translation (which includes some of the code in MORE, and all of the code in MORE2).

Regardless of the method of access, the particular locations are dedicated to fixed purposes and used in fixed ways. The label “parms” is referenced when BP is initialized for access to the entire data area, and the first two bytes of the area are used by the WRAPUP engine to access the address of its subpattern vector. The second two bytes contain a stack address to be fetched before executing a RETURN that allows an early (failure) exit to be taken from WRAPUP.

The sequence of declarations and a description of each is given below.

PARMS(4) Set and Referenced for WRAPUP (address of subpattern vector and SP = WRAPUP return).

P.XRF(4) is a pointer to the source (Index, Segment). It indicates where the appended cross reference table that identifies all of the externals for a source module begins. It is set by the state transition that occurs when a line of this type is matched. It is referenced by FINAL to append a new table.

D.USED(4) records the maximum value obtained by D.LIST when D.BASE indicates that Data Relative addresses are being allocated to data declarations. It is output as part of the .OBJ file to inform LINK how much space needs to be allocated for data within that module.

TOKEN(2) records the value (0-4095) of the most recently allocated token. If the WRAPUP function F.O detects a value > 4095, an “^” error is flagged for the statement.

LSAVE(1) contains the primitive code used to enter P.V and helps control the action of n’0A’ and EVAL.V.

STATE(1) contains the variable that is controlled by CK.STATE and that determines which statement types are valid or invalid at any point in the translation, and also whether the current statement is Global or Local.

C.MASK(2) records any value declared on an @PAD statement, and is passed to LINK. It controls the initial C.LIST value and the action of subsequent .PAD statements.

COLUMN.1(2) is set = FILE.PTR by NEXT.LINE when processing of a >prefix statement is not underway.

BODY.PTR(2) is set by P.Y during PREP, and by P.O during BODY. It is referenced by INIT.B.

OBJ.PTR(2) is set to OBJECT by INIT.P (when a PREP match is about to be invoked). It is used to compute the amount BMP.CODE adds to C.LIST (when called from NEXT.LINE), and by ERROR to reset OBJECT.

XRF.PTR(2) is set to XPATCH and is also used by BMP.CODE in computing amount added to C.LIST.

PREFIX(1), INDENT(1), and I.PREV(2) are used to define and control block structure.

L.NOW(2), L.PREV(2) record the length of the current and previous statement for use by NEXT.LINE.

C.BASE(2), C.LIST(4), D.BASE(2), and D.LIST(4) were discussed under *Symbols and Addresses*.

V.COUNT(2), VECTOR.0(2), and VECTOR.N(2) are counts and indices used by INIT.B to sequence through the set of alternative statement types given the first character of the current statement body.

FILE.BASE(4) is declared and set by the LF (load file) routine. It is not referenced again until SF (save file) is called to rewrite the source file to disk. It records the index and segment of the beginning of the source file.

FILE.PTR(4) is initialized by LF and is updated and used to record the beginning of the current line of source.

FILE.EOF(4) records end of source file. It is set by LF, updated by FF and FINAL, and referenced by SF.

F.HANDLE, DATE, and TIME are set and referenced by the file read and write routines (LF and SF).

FLAGS are used to communicate parameters to FF (fix file) during translation initialization. During MATCH translation, FLAGS communicate various conditions from one point in the MATCH-WRAPUP-NEXT.LINE cycle to another. The 2nd byte of FLAGS records the OR of any >prefix statements. The first byte is as follows:

```
      [abcd ef rr]       a: =5/6 or EmitTag; b: *pfx is OK; c: Label Error;
                         d: GLOBAL; e: NEXT; f: >prefix; rr: type of $ref.
```

MAX.SRC(2) and ERR.ADR(2) are used by backup and error reporting to diagnose an error at the right-most point of a match.

ERR.CNT (first byte only; second byte is a spare) counts the number of errors up to a maximum of 99 (BCD).

E.O.MEM(2) is the index to the top of the heap+256. If SP < E.O.MEM then a stack overflow has occurred.

TOKEN.PTR(2) is set and used by MATCH to rematch a previous token.

TOKEN.SEG(2) is the segment register used to access the 4K token buffer (MATCH output and WRAPUP input).

XPATCH(4) is the index and segment of up to 16K of Key:Token emits. All tokenized emits are recorded here, and C.LIST is incremented accordingly (by BMP.CODE in MORE2).

SYMTAB(4) is the index and segment of the beginning of the Symbol table (the entry above any root and load file entries).

OBJECT(4) is the index and segment of the next absolute byte of code to be emitted.

TEMP(16) is the LOOKUP buffer for symbol table access. Information is placed here to guide lookup and creation of new symbol table entries.

All of the above static data are accessible with a single-byte BP offset. An additional 26 bytes are present, but currently unused. Above the static data is the executable XPL code. Above that, a heap is grown upward toward the 64K segment limit, and from the top of the segment, a stack is grown downwards. Memory overflow occurs when SP becomes less than the heap pointer.

Starting with the next paragraph above the code (= stack) segment, is the source file. This may take either more or less than 64K. Above the source segment are the following segments: XPATCH (64K), SYMTAB (64K), TOKENS (4K), and OBJECT (64K). Although this sequence is completely independent from the use of each of these segments, during FINAL it becomes important. The object code is relocated to the top of its segment before it is combined with XPATCH and suffixed to the SYMTAB structure to produce the final format of the .OBJ file. This guarantees that the SymTab and Fixup areas can reach a full 64K each, without an overrun. Likewise, the now useless XPATCH area may be used to extend the source file with suffix lines recording all of its externals.

Within the 640K limits of DOS, there are 10 x 64K segments available. Allowing a couple for Debug or other overhead, and subtracting the 4+ taken up by all of the fixed XPL code and data, there is at least 200K available for the source file (at 28 characters per line, that’s over 7000 lines of source, or twice a “reasonable” maximum).

Block Handling

DEXPL allows two types of value references to be defined. Explicit references are defined by the use of the value primitives (x#x pattern elements). Implicit references are defined by the use of the LOOP, NEXT, and EXIT keywords in the "spec" part of a DEXPL definition. During MATCH, explicit references are handled by the P.V function, and implicit references are handled by the N.MOD function. Both of these functions require a unique token to represent the value they match. Explicit values are recorded in the symbol table. Implicit values are always code relative values, defined by block indentation. At any point in the translation of pass 1, each open level of indentation is represented by an entry on the heap. This entry contains the unique indent column and up to three tokens for the LOOP, NEXT, and EXIT addresses that may be referenced while that block is open.

The reference prefix of a statement gives the column number of any implict reference made by that statement. This column must equal some open block definition, or it must equal the body indent of the statement that makes it. In the latter case, N.MOD creates the block definition. Otherwise, the block definition must have already been put on the heap by N.MOD or NEXT.LINE as a result of an earlier statement. A failure to find an open block at the referenced indent column results in a "* prefix" error.

NEXT.LINE monitors changes in indentation (including >prefix statements), and puts new block definitions on the heap when indentation increases. When a reference to a block definition occurs, a token is allocated (if one has not already been assigned). Such tokens are initially recorded only in the heap and in XPATCH. NEXT.LINE puts special markers into XPATCH to define block points and labels. Labels and referenced block points are assigned tokens. The XPATCH entry that defines these tokens is a D:Token entry. Block points that are defined, but not (currently) referenced, are marked in XPATCH with a 9:<indent> entry. These entries only pertain to the currently open block. They are ignored if the block is closed without a reference being made to them. If a reference is made, the 9:<indent> is replaced by a D:Token.

Multiple Passes

XPL is designed to make multiple passes over the source. This is necessary to allow actual values to determine the course of a statement match. Often, the same syntax is used to define two statements whose only difference is that one uses a byte value, and the other a word or quad. In the absence of an explicit value or marker that indicates a word or quad, during the first pass over the source any unknown value is assumed to be a byte value. If these assumptions later prove wrong, additional passes may be invoked. Each pass represents the adjustment of one or more bytes to their word or quad defaults. This means the code address assignments must also be adjusted, and this further means that self-relative values may no longer be representable as byte integers.

During the first complete pass over the source, all self-relative addresses and value references that are not explicitly word or quad, are assigned byte values by default. When NEXT.LINE detects the end of source, it quits if there are any errors. Otherwise, it goes through XPATCH and finds all D:Token entries. These tokens are defined in the symbol table using D5 or D7 entries with decorated Keys. Because all symbols have now been defined, these additional entries will only be visible via token lookup. No equates to these entries are possible. Any 9:Token entries will be removed from XPATCH, but the D:Token entries remain in case of subsequent updates.

Then, the symbol table is scanned and all 7x-9x entries are evaluated and updated to type Ax-Bx, if possible. Next, XPATCH is scanned again and all Key = Byte entries are evaluated. If any of these are now defined as word or quad, another pass is triggered. This scan updates all the Keys in XPATCH that have been "promoted" so that references to them during the subsequent pass will get their correct sizes.

During all subsequent passes over the source, references are expected to exist in XPATCH, and are found using their "$" fields. After each pass, the same dual scan of XPATCH is done (unless ERR.CNT > 0). The result may be another pass, or closure. The only difference in passes after the first is that the scan for D:Token entries, resulting in symbol table updates, modifies the original symbol table entries—new entries are not required.

Final Processing

When a pass results in no more "byte promotions" and ERR.CNT = 0, two operations are performed by FINAL. The first of these is to suffix the Symbol table with the Fixup area according to the format of a .OBJ file. This is a merge of the OBJECT and XPATCH segments into the end of the SYMTAB segment, possibly overwriting the TOKEN and OBJECT segments. In preparation for this, the object code is moved to the top of the OBJECT segment so that an overrun is impossible.

The second phase of FINAL is creating the externals map and suffixing it to the source file. This is done using three pointers: The target for the X.MAP, a pointer to the source, and a pointer to the symbol table. The X.MAP target is determined by a pointer set when the old X.MAP is detected by the initial pass over the source. If none was present, the normal end of source is used. The symbol table pointer is used to scan through the symbol table looking for type 7x entries. The label found is searched for in the source. When found in the source, the first four characters of the line it is found on are copied to the X.MAP, then a space and the label itself are copied to the X.MAP. Four such pairs are separated by tabs, and each group of four is separated by CR, LF characters.

Details

This section contains design details including how values are matched by P.V and N.MOD, and how values are handled differently between pass 1 and subsequent passes. This section is initially a scratch design and will mature into changes to the above sections and the final design as implemented.

XPATCH

Entries in XPATCH are two words as follows:

```
        [ $:Delta ] [ Key:Token ]
```

Both words contain a 4-bit field followed by a 12-bit field. Key:Token (from a reference context) has already been defined. Where the least significant two bits of Key = 0, the entry corresponds to an Org Code. Otherwise, these bits define the size of the entry, or in the case of Key = 9 or D, a special “Define” tag is present. In all cases, however, the “Delta” field indicates the number of bytes of absolute code in OBJECT that logically precede the XPATCH entry. The “$” field is zero if it does not pertain to the entry. But, for each entry that contains a Key = 1, 2, 3, 5, 6, 7 (Byte, Word, Quad, Integer or Ordinal), or A, B, E, F (Word or Quad, Code or Data Relative), the “$” field indicates the MATCH token number corresponding to the final token of the entry. This enables the P.V and N.MOD routines, after pass 1, to get the Key:Token defined as a result of the previous pass, and use it to drive the logic of the current match. This is highly important in the case of block structure references, because it is only during pass 1 that heap entries are used to define the tokens for current block reference points.

Also note that, after pass 1, the “$” and “Token” fields of XPATCH entries do not change, but the “Delta” and “Key” fields may indeed change. This is because each pass emits the same alternating sequence of absolute and logical code. Each XPATCH entry is one-for-one with those of previous passes. However, the sequence followed by MATCH for each token it produces is not necessarily the sequence followed by WRAPUP for each token it emits to XPATCH. For this reason, it is important to record the “$” value (01-0E), of each token, so that a correspondence can be made.

The Symbol Table

Labels and constants are entered into the symbol table through the actions of P.V. When a label is first matched, a type 7x symbol table entry is created. NEXT.LINE processing detects if the label has appeared in column 6-7, and (during pass 1) promotes the label to a type 8x-Bx entry. If the label appears on any statement, other than a code-generating statement, the label entry will be fully defined with a decorated Key and a second entry will be made to define the value being assigned to that label. In the case of labels on code-generating statements, a D:<Def Token> tag is emitted into XPATCH, and definition is deferred until the end of pass 1, being done as part of the XPATCH scan that occurs after each pass.

The general rule for entries in the symbol table is that the Key of a type 4x-Cx entry (3x-6x and Cx being currently unused), gives what is known about the definition of that symbol. If Key = 0, the symbol is currently undefined (and after pass 1, is known to be an external). By pass 2, all defined labels (8x-Bx) are followed by a second entry containing the same decorated Key:Token as the label entry. This defines the value of the label. This entry may be an E5 (equate), or a Dx (definition) entry. Given a Token, the first occurrence of a Dx or E5 entry with that Token number contains the decorated Key of the definition of that Token. If no such entry exists for a given Token, the Token is currently undefined (and known to be external after pass 1).

When Tokens are assigned to block structure code points, they are defined similarly. Block structure Tokens are defined at the end of pass 1. A type D5 or D7 (based on PARMBYTE) entry is always made. No subsequent entries (E5, D6, or D8) are made that can reference these entries. Therefore, subsequent passes are free to update the values in these entries. It is possible that the values in these entries duplicate values already recorded in the symbol table, but lookup by value will always access the first occurrence of any value.

Block Structure

Three variables are maintained to guide the logic of indentation. These are the Indent column of any *pfx on the current statement, the indent column of the current statement’s body, and the indent column of the previous statement’s body. If the current statement has no *pfx, it is considered the same as the statement’s body. From one statement to the next, indentation can follow one of three patterns: It can stay the same, it can increase, or it can decrease. Normally, when it stays the same, no block is open at that level. When it increases, a block is opened at the previous level of indentation. When it decreases, blocks may be partly or completely closed.

Indent processing has three normal cases and two special cases. The normal cases are EQ, POS, and NEG indent. Here are the rules for defining new open blocks and closing currently open blocks.

1. The first indent (from zero) is considered to be an EQ indent (or an error if indent is < 9).
2. All EQ indents fall into three cases:
   a. No Block Opened: The current line contains no $ref to the current indent level.
   b. Block Opened: The current line contains a NEXT or EXIT $ref to the current indent level.
   c. Error: The current line contains a $ref to an erroneous indent level.
3. All POS indents fall into two cases:
   a. Block Opened (at previous indent level): The current line contains no erroneous $ref.
   b. Error: The current line does contain a $ref to other than an open, or the previous, indent level.
4. All NEG indents fall into two cases:
   a. The current statement is an EXIT statement: All blocks of equal or higher indents are closed.
   b. The current statement is a NEXT statement: All blocks of higher indents are closed, and the NEXT and LOOP references of the block at the current level of indent are redefined.
5. The end-of-source condition is handled as a NEG indent to the lowest open indent level (and current indent is set to that level). If no block indent is currently open, the current indent remains the same (to begin the next pass).

During pass 1, N.MOD is called upon to generate references to the local block structure. It normally must find a node already allocated that matches the referenced level of indent. If it does not, it will fail to set FLAGS(b) and a "* prefix" error will ensue from NEXT.LINE. However, by the above rules, it creates a node if a statement at the current level of indent has no *pfx and contains either a NEXT or EXIT (but not a LOOP) $ref. It also creates a node if the statement does have a $ref with an explicit *pfx to the previous level of indent. Otherwise, all new nodes are allocated by POS processing in DEF.BLOCK (called by NEXT.LINE).

Heap nodes are structured as follows:

```
        [indent] [loop token] [next token] [exit token] [ 8 ]
```

All of the token positions are set to zero when the node is allocated. Also, when the node is allocated, an entry to XPATCH is made to record the LOOP point of the block. This is done using a 9:<Def Indent> marker. When a token is allocated to replace a zero-word in the heap entry, the 9:<Def Indent> marker is replaced by a D:<Token> marker.

P.V and N.MOD

During all passes, P.V does a syntax match over the source corresponding to a x#x primitive. N.MOD is not invoked by MATCH until late in a statement match. Both functions need the capability of failing to match when a value is found to be out of range. This can occur immediately when a literal value is present. However, in all other cases, during pass 1, all unqualified values (those without a "pfx") are assumed to be within range, and during other passes, their sizes are obtained from XPATCH.

During pass 1, N.MOD uses tokens recorded in the heap to satisfy implicit references. During other passes, it finds the reference in XPATCH (by checking the "$" field of the next four entries).

LINK Design

Memory allocation = [OBJECT 64K] [AUX 64K] [XPATCH 64K] [.OBJ (i)   ], where each OJB(i) is processed back into object, aux, and xpatch, until no more .OBJ files remain.

Symbols & Expressions

Not withstanding the above, the following is a re-evaluation of what XPL currently does, with an eye to minor re-design and re-implementation and/or debug. The following may necessitate editing any of the above.

Symbols and expressions are used in a source language to designate Byte, Word, and Quad values that will be entered into the final binary object code. Expressions consist of symbols prefixed or separated by operators that evaluate (eventually) to simple binary values (Byte, Word, or Quad in size). The size of the value must be known at the time of the initial XPL translation. The actual value must be known at LINK time. Sometimes the size of the value cannot be determined when XPL first sees a symbol or expression. In this case, XPL assumes a Byte value, and later makes another pass over the source if this assumption proves to be wrong. If the assumption doesn't prove wrong until LINK time, a LINK error is produced (and an explicit "pfx" in the source is used to correct the error).

All symbols and expressions are defined with respect to the source by meta definitions (PTRAN for PREP, and DEXPL for standard code generating statements). Within the XPL translator, five data structures contain all the information about symbols and expressions. These are XPATCH, SYMTAB, and temporary entries on the HEAP to handle block structure references during the initial pass over the source, temporary entries into TOKEN.SEG to pass MATCH tokens to WRAPUP, and temporary entries into OBJECT to pass the results of a sub-match and its WRAPUP back into the primary MATCH. During any secondary passes, references are primarily to XPATCH and SYMTAB. References to the HEAP do not occur during secondary passes, and other primitives and functions may be revectored to simplify the actions they have to perform after the first pass.

During all passes over the source, the MATCH and WRAPUP engines are driven by the same meta definitions. They use the same logic at the highest level. Minor differences exist in the logic of the lowest level primitives and in the processing that occurs between statements in NEXT.LINE. Let's follow the logic of the recognition and code generation of XPL values in a top-down fashion.

Meta-Definitions

At the highest level, values are defined in PTRAN and DEXPL. There are 15 different primitives that match XPL values. The first 12 are allowed in DEXPL definitions, and all 15 may be used in PTRAN definitions. When a value is matched, the source may specify the value literally and/or it may use a Word or Quad prefix to specify the value. If the value is ambiguous in the immediate context of the source, a Byte value will be assumed during the initial pass, and an actual value (if known) will be substituted in subsequent passes. This process begins when any of the primitives (a, b, .. k, l, v, w, or z) is used in a PTRAN definition, or when either .# thru ~## (equivalent to the 12 primitives a, b, .. k, l), or an implicit reference, is used in a DEXPL definition.

Typically, the primitives a, b, .. k, l are all funelled through the primitive v, and only the primitives w (for a quoted string) and z (for an alpha.numeric label) are handled in the normal direct fashion. When the primitive v is invoked, a deeper level of MATCH and WRAPUP occurs. When v is referenced directly, it matches only a single symbol. When it is invoked via one of the 12 DEXPL primitives (a, b, .. k, l), it matches the extended syntax of an optional unary prefix operator and/or binary operator with a second symbol operand. DEXPL also allows implicit reference to values. This occurs in references to the block structure, using the DEXPL "spec" syntax, which is translated into instances of the n primitive (P.N calls N.MOD to decide the match and emit the token).

Symbol & Expression Matching

Matching the source (regardless of the pass) begins with matching the "prep" (PTRAN) pattern definition. This definition has 11 alternatives. A 12th alternative is generated when all of the prep alternatives fail and a DEXPL alternative succeeds. Since all DEXPL statements begin with an optional label, but DEXPL does not address the definition of the label field, label definition is handled with the "prep" MATCH and the "back" WRAPUP. When either "prep" or DEXPL patterns are matched, the primitives (a, b, .. k, l, v, w, and z) are called to match values. The first 13 of these invoke a sub-match and WRAPUP on the general syntax for a value reference. This results in a temporary 3-byte entry at the end of OBJECT (which must be erased). The information in this entry gives the form (leading unary or not), and (known or defaulted) the type (integer or ordinal, and Byte, Word, or Quad). This is verified against the calling primitive and success or failure is taken as a result. Success results in a token being emitted in the higher level MATCH, and may result in the higher level WRAPUP making an XPATCH entry.

Label definition occurs during the first pass and is handled by the initial WRAPUP. Code labels result in a special XPATCH entry that enables the label to be defined after each pass is complete. The WRAPUP functions, F.X, F.Y, and F.Z are responsible for label definitions in the symbol table. The F.X function handles CODE= and DATA= declarations. F.Y handles <label> (v) and <label> = <expr> declarations. The F.Z function handles a label on a code statement. All of these require only the initial pass to do their jobs. In subsequent passes they are revectored to code that merely performs housekeeping. All differences between passes occur as a result of MATCH failing on values that were Byte in the previous pass, and that are Word or Quad in the next pass. This causes new addresses to be assigned to code statements, and therefore different values for labels and expressions, but it does not require that any label or expression definitions be made more than once.

All values are emitted as token entries into XPATCH. The Symbol table is used to record the binding between a token and its value. Values may be written directly using a literal with appropriate prefixes or a unary operator, or indirectly using a label. The binding between a label and its value requires two symbol table entries: The first binds a token to the label, the second binds the token to a value. There are ten types of values: Six are absolute (Integer or Ordinal, and Byte, Word, or Quad), two are Code Relative (Word or Quad), and finally two are Data Relative (Word or Quad). All literal values are absolute. Values expressed with one or more labels may be (known or assumed to be) any of the ten types.

Now, consider how values are represented in DEXPL and PTRAN. Fifteen primitives are devoted to the various ways they may be written. Twelve of these also direct the type of value required. These primitives are designated in three different ways. In DEXPL the primary 12 are written with the symbols [.# +# –# ~#] for the Byte primitives, [.#@ +#@ –#@ ~#@] for the Word primitives, and [.## +## –## ~##] for the Quad primitives. Each set of four indicating Ordinal, Integer (with required + or – addop), Integer (with optional + or – addop), or a self-relative integer (no addop allowed). In PTRAN the primary 12 are referenced by the letters [a .. l] and the secondary three by the letters [v, w, and z]. Inside the XPL translator, the binary codes are [0C2, 0C4, .. 0D8, and 0EC]. The codes for the letters [w and z] are [0F0 and 0F4], but they are unimportant. These bindings are part of the XPL design. The v primitive indicates a single symbol. It may be a label or any of the forms of literal. The w primitive indicates a quoted string. The z primitive indicates a label. None of these three implies any restriction on type or size (except that v and z may not represent a size larger than Quad).

When a value matches a primitive, the result is success or failure. If the result is success, WRAPUP may emit a Key:Token into XPATCH. In the case of all but the w and z primitives, symbol table entries are made. The w and z primitives merely match the syntax of a string or label (emitting an intermediate token pointing to the source string). Symbol table entries are made as a result of the sub-match invoked by the primitives [a .. l, and v]. When the sub-match is invoked, the ID of the calling primitive [0C2 .. 0D8, 0EC] is passed to it. The sub-match succeeds only if the syntax required by the calling primitive matches that actually present. That means that the 0C4 alternative may be found only if requested by one of the [0C4, 0C6, 0CC, 0CE, 0D4, 0D6] primitives. The 0C2 alternative may be found only if requested by any of the primitives *except* the [0C4, 0CC, 0D4] primitives. The v primitive matches the 0C2 (symbol only) alternative. In addition, the sub-match returns a Key:Token that defines the value matched, and guarantees that this Key also matches the requesting primitive. The Key is computed from what is known (at that point) about the syntax and values of the operands and operators matched. If there is not enough known about the value being matched, the Key will be returned as zero, and Success will be assumed.

The symbol table is used to record every label, value, and expression. The first appearance of a token in the symbol table gives its “base.” A token’s “base” can be External, Absolute, Code Relative, or Data Relative. An external base is indicated by a Key = 0. Keys = [1, 2, 3, 5, 6, 7] indicate Absolute (and Byte, Word, or Quad, and Ordinal or Integer). Keys = [A, B, E, F] indicate Code Relative or Data Relative (and Word or Quad).

Implicit values may also be defined in DEXPL. These values reference the program’s block structure using a “def = mod [,mod] [,mod]” where at least one of the “mods” is a [LOOP, NEXT, EXIT] keyword, and one of the keywords [ORD, INT, REL] is combined (explicitly or by default) with one of the keywords [BYTE, WORD, QUAD]. The defaults are [REL, BYTE]. The REL attribute means “self relative.” The Key for this type of value always indicates a Byte [~#], Word [~#@], or Quad [~##] Integer. The Token emitted into XPATCH matches a Symbol table entry that records a unary (self-relative) operation on a Token recorded elsewhere in the symbol table. It is the first appearance in the symbol table of this second token that records the “base” of the relative target.

The first appearance of a token bound to a label is always the symbol table entry that contains the label. The second appearance of the token may bind it to a value, an expression, or an equate to some other token. Values recorded in symbol table entries [D4, D5, D7] are absolute ordinals. Their Keys are typically [1,2,3], because they typically do not contain the first appearance of their tokens (the initial entry is the one that specifies the base). However, when a value is bound implicitly to a token, the [D4, D5, D7] entry may be the first symbol table entry containing that token. In this case, its Key may indicate a Code or Data relative base. The other possibility is that the implicit value is already recorded in the symbol table. In this case, the new token is entered into an equate with its Key indicating its base, and referencing an earlier token with a different base.

Value Processing

Values are written into the source and matched with PTRAN and DEXPL pattern definitions. This match interprets the value, represents it with entries in the symbol table, and WRAPUP emits it into XPATCH and ultimately into the object. Let’s follow the processing that occurs in the MATCH and WRAPUP of values in various XPL statement types.

The simplest MATCH processing occurs when literals are matched to the (PTRAN) “prep” and “match.prim” definitions. These literals are various forms of binary, hex, decimal, and string constants. Certain “fix” and “pfx” operations are optional. MATCH hands these literals to WRAPUP in the form of a sequence of options and alternatives that it has matched. A WRAPUP function does the actual interpretation and records the result.

The simplest case is the “pad” pattern in “prep.” This matches 1-5 hex digits and is interpreted by the p WRAPUP function. Information is recorded in C.BASE, C.LIST, C.MASK, and XPATCH. In “match.prim” the entire syntax of XPL values is defined using only two primitive functions (for convenience), w to match a quoted string, and z to match an alpha.numeric label. The MATCH result of a match with “match.prim” is a sequence of options, alternatives, and pointers to source strings. A successful match is handled by the u WRAPUP function (preceded by a byte parameter with one of the 8 values [0, 2, .. C, E] to indicate which form of symbol is to be handled). Each invocation of u converts the source string into a value or label and enters it into the symbol table. The syntax of the match, the token of the symbol table entry, and any explicit size or type information are emitted into OBJECT.

The MATCH of “match.prim” and subsequent WRAPUP of “wrap.prim” are invoked by the MATCH of any of the 13 primitives [a, b, .. k, l, and v]. Each of these specifies an allowable syntax, type, and size of value. The result (in OBJECT) is verified by the caller, and an intermediate token is emitted by the higher level MATCH. This token may result in a final token being emitted to XPATCH (for a DEXPL WRAPUP), or some other action (for a WRAPUP of “back”) involving the WRAPUP functions [v, w, x, y, and z].

The Use of Keys in [Key:Token]

The Key:Token (a 16-bit Word) is used in the following places: In the C.BASE and D.BASE variables, in the primary position of a symbol table entry, in a secondary position of a symbol table entry, in an Xpatch entry, and temporarily in Object to pass the results of a value sub-match back to a higher level match. A brief explanation of the Key code is given on page 22. Here is further elaboration.

C.BASE and D.BASE

Key = 0 (Token > 0) indicates the Code or Data Base is an external (with an offset of zero). Any initial or current offset is indicated in C.LIST(4) or D.LIST(4).

Key = 4 indicates the Code or Data Base is absolute with any offset recorded in C.LIST(4) or D.LIST(4).

Key = [8, C] means the Code or Data Base is Code or Data Relative, respectively, and any offset is indicated in C.LIST(4) or D.LIST(4).

When Key > 0, Token = 0. No other Keys are used.

OBJECT

The purpose of a Key:Token in the temporary Object is to indicate what is known about the value whose token it is attached to. Note: Token > 0, always. Only the following Keys are used.

Key = 0 indicates External (or forward reference), Byte assumed.

Key = [1, 2, 3] indicates known (Byte, Word, Quad), Ordinal assumed (local or external).

Key = [5, 6, 7] indicates known (Byte, Word, Quad), Integer (local or external).

Key = [A, B] indicates local (Word, Quad) Code Relative.

Key = [E, F] indicates local (Word, Quad) Data Relative.

Symbol Table

Key = 0 in all undefined label type symbol table entries (6x, 7x, 8x, and 9x entry types).

Key = 8 indicates Code Relative in Ax, Bx entry types (and D4, D5, D7 entries made by SCAN.1).

Key = C indicates Data Relative in Ax, Bx entry types (and D4, D5, D7 entries made by SCAN.1).

Key is copied from Object (see above), into the secondary positions of all D6 and D8 entry types.

Key = 0 (typically, exception noted above) in the primary position of all Dx entry types.

XPATCH

Key = 0 means “Set the Code Origin to the external = Token.” If Token indicates a binary expression and the value of the second operand is absolute (a simple, known offset), C.BASE is set to the 00:Token of the first operand, and the value of the second operand is loaded into C.LIST(4).

Key > 0 is exactly as noted on page 22, and more specifically, as follows:

For Keys = [4, 8, C], the Code Origin is set to the appropriate base and the value of Token is used to set C.LIST(4).

For Keys = [1, 2, 3, 5, 6, 7, A, B, E, F] a Byte, Word, or Quad value is to be merged at that point with the Object.

For Key = 9, the head of a block (at indent in place of Token) is defined at that point in the Object.

For Key = D, a Token is defined at that point in the Object.

AA.VALUE

This function is called with AA = X:Token, and it returns BC:HL = value, AA = Key:000, where:

Key = 0 if the value cannot be determined.

Key = [1, 2, 3, 5, 6, 7, A, B, E, F] if the value, size, and base are determined.

Symbol Table Management

All entries into the symbol table are made with a call to ACCESS. Parameters are set up in TEMP(16) before the call is made. TEMP bytes are as follows:

TEMP+0: The length of the string (1-12) to be accessed.

TEMP+1: The type of the entry (40-DF, typically 60, 70, D0). Entry types 00-3F and E0-FF are invisible via ACCESS. Entry types 40-BF are distinct (even if their string parts are identical) from types C0-DF.

TEMP+2 (12): The string part of the entry (alpha.numeric label, binary constant, or coded expression).

TEMP+14 (2): A Token "request" (second subtlety below).

ACCESS has two subtleties:

1. When type = 070, a new entry causes column 1 of the current line of source to be "marked." The first line in which a label forward reference appears has bit 7 of its first byte set. This is used by X.LIST.

2. When a Token "request" is made, a new entry will be defined with the requested Key:Token. Otherwise ("request" = 0), the next Token value in sequence is assigned, and Key = 0. However, if a new entry is not required (the string is already defined), an Equate entry is made, equating the requested Token to the Token of the existing entry.

Note to Self:

A self-relative reference must fall into the following classes:

1. A direct reference to an absolute integer value means use the value as it stands.

2. A direct reference to a point in the local code means compute an integer from the difference between the reference and the target.

3. Any reference to an external or absolute ordinal means wait until LINK time when the reference address is known and then compute an integer from the difference between the reference and the target.

Routines Involving Symbols

P.A – P.L

These primitives correspond to the 12 "value" patterns in DEXPL (indicated by x#y, where x = [. + – ~], and y = [nil @ #] to indicate the operator syntax allowed and specify [Byte], Word, or Quad). Each primitive invokes a MATCH of <match.prim> and WRAPUP of <wrap.prim> which can also involve P.N, P.W, P.Z, and F.U. Each primitive then does the checking and further processing, unique to the values and syntax allowed for that primitive, before emitting the appropriate Match Token.

For the ~ (self relative) primitives, an Ordinal reference (unsigned) may be to a Code Relative Target, or to an Absolute value. In either case, a special unary expression indicating "self relative" must be substituted in place of the direct reference. However, in the case of an Integer reference (explicitly signed, and to a non-Code Relative Target), the direct reference may be emitted.

All cases, plus N.MOD (below), which is specifically concerned with Code Relative Targets, involve emitting a Match Token containing codes that represent the requested primitive, the canonical form actually matched, and a Key:Token to be emitted into the Object by Wrapup. The actual match and Key must conform to each other and to the requested primitive, otherwise a Match Failure (CY=1) is returned.

These twelve primitives are combined into six groups: P.A handles both Ordinal and Integer Bytes, P.D handles Self-Relative Bytes, P.E handles Ordinal and Integer Words, P.H handles Self-Relative Words, P.I handles Ordinal and Integer Quads, and P.L handles Self-Relative Quads. During Pass = 0, only the syntax determines the size of a value. During Pass > 0, the Key in XPATCH (updated before the current Pass) may be used to "promote" the size. Then, requested type and size are compared to actual size and type, and Match Success or Failure is taken accordingly.

P.N

This is a multi-purpose primitive. It gets parameters from the pattern bytes that immediately follow it. The only parameters that pertain to Symbol matching are: n'0A', n'0B', and n'1x'. The first two of these are only invoked from a sub-match. n'0A' fails only if the sub-match was invoked from the "v" primitive (it succeeds if any of the primitives "a" – "l" invoked the sub-match. This is how the full expression syntax is matched by the 12 primitives, and the limited syntax of a "symbol" is matched only by the "v" primitive.

The n'0B' primitive merely insures that any digit string (decimal or hex) does not stop short of an additional digit, if one was erroneously present in the source.

The n'1x' primitive is much more complex than the above. The 'x' field specifies a $ref (a Match Token position), and a second parameter byte gives a list of keywords that are to be applied in generating an intermediate Match Token. If the keyword list omits LOOP, NEXT, or EXIT, the n'1x' action merely checks that the Token already produced corresponds to the other keywords present [BYTE/WORD/QUAD and/or ORD/INT/REL]. If a block structure keyword [LOOP, NEXT, or EXIT] is present, the n'1x' primitive, itself, produces the intermediate Match Token (position given by the $ref field) according to the other keywords present together with the indent point of the statement's reference prefix. During the initial pass, this means accessing a HEAP entry to get the appropriate token, and emitting this token with the requested key. During subsequent passes, it means accessing XPATCH to find the next token to be emitted to the $ref token position, and using the Key:Token found there. If the pattern attributes don't match the key that is found, the n'1x' pattern element fails.

P.V

This primitive is used in several places in the definition of "prep." P.V is simply vectored to P.I, invoking a sub-match on "match.prim" and wrapup on "wrap.prim." However, it matches a simple, unsigned, value whose size and type are either dependent on context or may be Quad. If the sub-match succeeds syntactically, a token is emitted, and WRAPUP will determine if the type or size were in error.

P.W

This primitive matches a quoted ASCII string of any length. It is always matched after a hex string (in single quotes) has had a chance to match. Its results are handled by two different WRAPUP primitives, depending on the context from which it was referenced. When handled by F.U (always as an earlier MATCH alternative), the string length is limited to Byte, Word, or Quad. When handled by F.S, the string may be of any length (other than Byte, Word, or Quad, if those possibilities have been removed by matching an earlier alternative).

P.Z

This primitive merely calls the BUFFER routine to match an alpha.numeric label. No symbol table entry is made, and the intermediate token is simply a handle on the source string.

F.A – F.N

These 14 WRAPUP functions merely access their respective intermediate tokens (by position), and emit a corresponding XPATCH entry. The Key:Token is transferred without change. The $:delta part of the XPATCH entry is computed based on the token position and the number of bytes of code emitted to OBJECT since the previous emit to XPATCH.

F.O

This WRAPUP function handles State Change. It insures that the proper statement sequence is followed.

F.P

This WRAPUP function handles .Pad= and @Pad= statements, producing Org Codes as appropriate.

F.Q

This WRAPUP function handles the Name=file statement.

F.R

This WRAPUP function handles the "[v *] value" alternative of "string" in the "prep" definition of a code emit declaration statement. If "v" is omitted, it defaults to one; otherwise, it must be defined within a range that does not cause overflow. It is used to repeat the "value" part, which must be a Byte (by default), Word, or Quad.

F.S

This WRAPUP function access the ASCII string matched by the “w” primitive and emits it to OBJECT. In doing so, it applies to each character any bit toggling indicated by the optional “fix” operator.

F.T

This WRAPUP function handles the Load=file statement.

F.U

This is a multi-purpose WRAPUP function. It is only referenced from the “wrap.prim” definition. It gets a parameter from the byte immediately preceding it in the WRAPUP data structure (one of the eight values 00, 02, .. 0E). The first six parameter values simply convert intermediate tokens to <label>, binary, decimal, hex, ‘hex’, or “string” values (Byte, Word, or Quad, as appropriate). The binary constant is entered into the symbol table, and its token is emitted to OBJECT. The final two parameters direct the handling of (respectively) a binary or unary expression,. The result of this WRAPUP is a 3-byte entry in OBJECT (that must be erased) that gives the syntactic alternative of “match.prim” that matched, and the Key:Token that corresponds to the overall expression or symbol.

F.V

This WRAPUP function is only referenced from the “back” definition to handle the token emitted when “v” is matched as a part of the “prep” “primary” pattern. It merely moves the Key:Token from the intermediate token into a 5-byte packet emitted into OBJECT by the “back” WRAPUP. By checking the presence of a binary operator, it selects the left or right side of the packet for inserting the token.

F.W

This WRAPUP function does for “expr” in a “primary” what F.V does for “v.” In this case, it must build a symbol table entry for the expression from OBJECT, collapse OBJECT by 5 bytes, and put the Key:Token into the previous entry in OBJECT. Again, this result is put into the left or right part of the packet, depending on whether a binary operator is flagged.

F.X

This WRAPUP function is called when an expression has been fully handled, and the final 5-byte packet is in OBJECT. The CODE= or DATA= statements are finalized. CODE= results in an XPATCH entry that signals a new Code Base and/or offset. DATA= sets a new D.BASE and D.LIST. The OBJECT packet is erased.

F.Y

This function handles the WRAPUP of two further syntaxes: “<label> (v)” and “<label> = <expr>”. These statement types define a <label> in terms of D.BASE or an explicit <expr>, and the former also updates the value contained in D.LIST. Again, the OBJECT packet is erased.

F.Z

This function defines a <label> in terms of the current C.BASE and C.LIST. It is invoked as part of the WRAPUP of any code emitting statement (initial alternative = 040 or 09, a normal XPL statement, or a “[<label>] [:] = <string> [, <string>]..” statement.

NEXT.LINE Processing

Between one XPL statement and the next, block structure processing takes place during the first pass. During all passes label and prefix error checking takes place. If the statement is not a >prefix statement, the next line of source is accessed in conjunction with an end-of-file check. When end-of-file is detected, CK.PASS2 is called for post-pass processing and a decision as to whether another pass is required. NEXT.LINE returns CY if translation is complete, otherwise it sets up the pointers to the next point in the source.

DEF.BLOCK

Called by NEXT.LINE only during the first pass, this routine updates the length and indent points (prefix and body) of the previous and current statements. It then calls a routine appropriate to the indent increasing, decreasing, or remaining the same. A heap entry is created for each open block, block reference points are defined, and open references are defined. Blocks that are closed are deallocated.

Left Recursion

The following grammar needs to be implemented via PTRAN and OTRAN:

```
expr→     seq
          unary + seq
          unary - seq
          unary * mult
          unary / mult
          unary ** mult
unary→    prim
          + prim
          - prim
seq→      mult                    seq =   mult [a.op mult]..
          seq + mult              a.op =  + | -
          seq - mult
mult→     expo                    mult =  expo [m.op expo]..
          mult * expo             m.op =  * | /
          mult / expo
expo→     prim
          prim ** expo
prim→     label
          constant
          (expr)
```

to demonstrate that the power of the XPL Match Engine is sufficient to handle standard parse trees, and a left recursive parse in particular. The above is intended to be as general as the Fortran definition of an expression. If it is not, it should be debugged (for example, it should evaluate the following in the correct sequence: $–a+b^2–4ac$).

11/24/01The current Match engine is really not good for left recursive grammars, and using the repeat construct doesn't help a lot. The trouble is that the search for the correct parsing can get quite time consuming with the current algorithm for backup and retry. I've been giving a lot of thought to a redesign of PTRAN and the Match algorithm to better handle more universal grammars and be more efficient at run time. There is currently a bug in DEXPL, which causes it to create a bad pattern. Likewise, the |= $BP pattern that is supposed to be the equivalent of $(BP+0), probably doesn't work in the current design. I can fix both of these, and should, and complete XPL along the current design lines, but here I wish to note the thinking I've done about the more general design, and one that should execute faster than the current one, as well.

First, PTRAN would be extended to cover the generality of DEXPL (its algorithms would be used to implement a new version of DEXPL). The new MATCH would replace all the current MATCHes. The MATCH logic would be as follows:

```
CALL INIT.F                             external
DO      CALL INIT.P                     external
        CALL MATCH
        IF CY,
                CALL INIT.B             external
                CALL MATCH
                IF CY,
                        CALL ERROR      external
        *               EXIT
        ELSE    CALL INIT.T             external
                CALL WRAPUP
        CALL NEXT.LINE                  external
LOOP UNTIL CY
CALL FINAL                              external
```

Both MATCH and PTRAN would distinguish two types of grammars: Limited and Full. A limited grammar would involve fixed position tokens only, and therefore have no brackets or recursion. There would be three code generating statement types to PTRAN. Pattern definitions, statement definitions, and code emit definitions. Thus, a source program could look like either a current PTRAN source file, or it could look somewhat like a DEXPL file, however, a mixture of code generating statements with a Full grammar would be diagnosed as an error.

Any Full grammar would have to be paired with a code emit definition prepared using OTRAN. For a Limited grammar, it would be optional whether the code emits were embedded in the grammar, or prepared separately using

OTRAN. The general rules for PTRAN source would be, statement definitions would be separated from other lines with a blank line or comment. No continued statement could have interleaved comments, except that two+ spaces followed by a "." would signal a comment and end of line terminator. The Limited Grammar Data Structure would be as follows:

```
      @B.Vect, @Sub.1, @Sub.2, ..
      [B.vect:]  N1, B1, @Pat1.1, @Pat1.2, ..
                         N2, B2, @Pat2.1, @Pat2.2, ..
                         ..
      [Pat1.1:]  ^B.1, B.2, .. ^P.1, <pattern> [^Next]
                         ..
```

Where @Sub would address subpatterns, and @Pat would address statement definitionss, but both would have the form of the <pattern> line, a two or three part sequence giving a beginner list for the pattern, then the pattern itself. ^Next is either a code emit definition, or the next ^Beginner list.

A Full grammar would have the following form:

```
      @Pat.1, .. @Sub.1, @Sub.2, ..
      [Pat.1:]           ^B.1, B.2, .. ^P.1, <pattern> [^Next]
                         ..
      [Sub.1:]           ^B.1, B.2, .. ^S.1, <pattern> [^Next]
                         ^B.1, B.2, .. ^S.2, <pattern> [^Next]
                         ..
```

Where, zero or more root patterns are defined by statement definitions, and a set of sub-patterns are defined by regular pattern definitions.

In both of these definitions the sequence of ^B, B, .. indicates a list of beginners, one of which must match for that (sub) pattern to match. In the Limited Grammar, the N, B vectors are used to list all of the statement patterns that have a given beginner. When Match is invoked, the following register setup is required:

```
                 BP=@N.B.Vector (or @B.B.List)
                 DE=@Sub.Vector
                 DS,SI=@Source
                 ES,DI=@Object
                 BC (is used internally by Match)
                 AA,HL = Scratch
```

The key to these conventions is that the m.s. bit signifies what you are looking at when a substructure begins.

During a Match, the Token Buffer would contain 4 types of entry:

```
      [ 0 ] [ e ] [@Source]      error code at right-most position
      [len] [alt] [@Source]      len=1-96, for literal match
      [prim] [len] [@Token]      prim=97-122
      [sub] [alt] [@Object]      sub=128-255
```

Match would match a beginner before going further with any pattern, it would detect left recursion, and it would keep the longest matched sequence of tokens. Any backup retry would be recorded tentatively, until the match were extended further into the source. Left recursion would be recorded by inserting a [Sub] [Alt] entry as necessary for each recursion.

XPL source files (PTRAN, OTRAN, DEXPL, XPL, other):

(1) Line separation = CR, LF, or CR, or LF.

(2) Abort if any line > 255 characters, or begins with a character code 128-255.

(3) Abort if any line contains lower case alpha within the first 5 characters, unless it is a comment or equate.

(4) Any line shorter than 7 characters, beginning with a "." or "<<", or containing a ":" as one of the first 6 characters is a comment and will be preserved unchanged.

(5) 3-5 characters will be written into the beginning of any line in error. The sequence "?=" is always part of the error message (although, a natural "=" sign is used, in the case of an equate statement). As a prefix to this, the column number "nn" of the error is inserted.

(6) An equate is any line with an "=" sign in column 5 or 6. An equate is defined by its first character and the presence of the "=" sign. A default first character is assumed. Columns to the right of its "=" sign are preserved.

(7) Columns 1-5 may be overwritten in any other form of line, all other columns are preserved.

Canonical Reshaping **(insert in main doc if and where appropriate)**

DEXPL statement definitions may have several forms that need to be put into a canonical form before they can be expected to work correctly in a standard pattern Match and Wrapup. After the object code for a statement has been emitted, the following sequence of calls is made by F.0 (indicated by ` in the Otran definition): CALL CK.LEFT, SORT.ALTS, F.1 (FIX.MACRO, STUFF, FIX.SHAPE, CRUSH). The first two of these, CK.LEFT and SORT.ALTS, perform the same function for DEXPL as for Ptran. They reorder the alternatives of a pattern to make proper heads come last (and put in explicit Alt Overrides as necessary to retain the original Alt numbers).

Fix.Macro builds a statement definition on top of the subpatterns which constitute the sequence of alternatives of the macro definition. The final statement definition is simply a sequence of references to the parts of the macro, separated by references to the "o" primitive, which handles continuation from one line of the macro to the next.

Once these operations have been performed on the object data structure, Stuff, Fix.Shape, and Crush are called to put the object into its final canonical form. Stuff and Crush are easy to understand. Stuff inserts a (redundant) Alt # override in front of each subpattern and primitive pattern element (in the alternatives of a main statement definition, not into any of the subpattern definitions). Crush removes any of these Alt # overrides that are still redundant after Fix.Shape has done its work.

The purpose of Fix.Shape is to insure that a pattern Match of any statement alternative always produces the same sequence of tokens. If $2 refers to a 3-bit field in one alternative, it should refer to a 3-bit field in the others as well. If $3 refers to a # primitive in one alternative, it should do so in all alternatives. Fix.Shape does repair the pattern data structure in two situations: If one Match sequence omits a # reference (in a subpattern) that is contained in another sequence, an element override in the main statement definition is inserted to force the pattern element after this subpattern reference to a fixed value. Secondly, Fix.Shape detects a statement definition with alternatives in a different sequence. It insures (for certain cases, using overrides) that equivalent elements of each alternative have the same numbers. Finally, Fix.Shape detects errors in the sequence of alternative Match elements (where $n might be defined by a 2-bit field in one Match sequence, and a 3-bit field in another sequence). Every Match needs to produce the same sequence of field sizes and/or calls to a # primitive.

Fix.Shape invokes Do.Shape when it encounters an element # override. Before the first alternative, D.ALT is set to ^0. Do.Shape returns immediately if called with the first alternative. It is only interested in the second and subsequent alternatives. The D-register is set to the actual element following the override (there may be 2 overrides in sequence). Then, starting with the Main Alternative, each previous alternative is scanned (possibly running into the current alternative). When a subpattern or #primitive is encountered, various things are checked. If it's the first element, the previous and current alternatives are compared. If they are identical (#prim = #prim, or the same subpatterns), then D.ALT will be set zero, Do.Shape will be exited using the current override to replace itself, and the next entry to Do.Shape will detect D.ALT = 0, and exit without doing anything. If they aren't identical, it can be two different subpatterns (D.ALT < 128), or a #primitive and a subpattern (D.ALT > 128). When D.ALT is negative, if the current element is a subpattern, the first subpattern found satisfies the search, likewise, if we are looking for a #primitive, the first #primitive found satisfies the search. However, if D.ALT is positive, the first #primitive found satisfies the search for a #primitive, but only an identical subpattern satisfies the search for a subpattern. Notice that we are guaranteed satisfaction, because if the search goes far enough, it will arrive at the current element and match it exactly. This algorithm doesn't reshape all possible definitions, but it does handle the ones in the current version of XPL, and they serve as examples of this capability.

STRIPE (STRIng Processing Environment)

This would be a programming language and a compiler for it, somewhat similar to SNOBOL. The compiler would generate data structures, emit .OBJ modules, and invoke LINK (all from within itself). It would delete any temporary files, and leave the source directory with a .COM file as the result.

A program consists of pattern definitions and executable statements. Pattern definitions have a syntax and semantics similar to PTRAN, with four exceptions: There are no "head overrides," parentheses may surround a sequence of pattern elements, the → operator and a <word> may follow any pattern element or grouped sequence of elements, and a pattern element is defined below as the <source> construct.

Unquoted literals are matched without regard to single spaces, as in PTRAN, but quoted literals must match exactly. Uppercase alpha in an unquoted literal matches either upper or lower case. Concatenation of elements may be indicated with an intervening null (if syntactically separated otherwise), or a space, and have the meaning that an intervening space will match optionally, but not be generated. To prevent a space from matching, an intervening "~" must be used as a concatenation operator. To generate a space on output, it must appear in a quoted literal (or be contained in a stored string).

The syntax of a statement is:

```
[<label>:] <body> [S:<label>] [F:<label>]
```

A <body> may be either of the following:

```
<source> <relation> <source>
<source> [: <pattern label>] [→ <object>]
```

A <relation> may be any of the following:

```
< | <= | > | >= | <> | =
```

A <pattern label> is a name that labels a pattern definition.

An <object> is either the name of an object function (pre-defined), or the name of a memory location.

A <source> may be a <literal>, a word, an indirect, a redirect, an ordinal, or the name of a source function. A <literal> is quoted or unquoted as explained above. A word is the name of a memory location in which a string may be stored. An indirect is indicated as $<source>, and a redirect is indicated as @<source>. An ordinal is a string of digits with no other characters. The + and – operators exist to combine two ordinals, giving a sum or difference. If the result is not an ordinal, the result is null. A special interpretation is given to $<ordinal>, where <ordinal> is an unquoted literal. The value is always another ordinal, defined by the most recent pattern match, where the <ordinal> gives the order of a subpattern in a match; the value returned is the number of the alternative that matched (just like the similar syntax in PTRAN/OTRAN/DEXPL). Any other use of <ordinal> is just like any other string.

A <source> without an @ or $ prefix refers to a literal, or a stored or generated string of ASCII characters (excluding 00 and FF). Strings are normally stored as the contents of a word. The word "names" the contents. However, the $ prefix allows one string to be used as the name of another stored string. The @ prefix works this in reverse: It returns the (first) string that names a given content. Either of these operators may return null.

Example: abc contains xyz. If xyz contains 123, $abc = 123, and @123 = xyz.

This is actually implemented as follows. Words and literals are contained in a hard-coded symbol table. When any string is defined with contents, it is entered into a dynamic symbol table as a (name, content) pair. All names in the symbol table are unique. A name's contents may be redefined. If a name is defined as null, it is simply deleted. A given content string may appear several times in the symbol table.

Only one source and one object function are guaranteed to be present: "input" is the standard input file, and "output" is the standard output file. When invoking a STRIPE program, these may be defined with the redirect characters to named disk files, otherwise they default to the computer's keyboard and display.

Standard input is generally "filtered" to the extent that HT characters are expanded to spaces in columns of eight, and any CR [LF] or [CR] LF is turned into a single LF character. In patterns, the characters HT and CR may be used to indicate 1+ spaces and 1+ line endings. To standard output, the desired characters must be indicated exactly.

Notice that statements whose bodies don't invoke a pattern match or an assignment, can only result in success or failure. A simple <source> body results in failure under one of two conditions: If the entire <source> is null (a source function returns null), or if a null is produced in place of an ordinal. A comparison between ordinals is numeric, between any other strings, it is lexical.

Statements that invoke a pattern match or assignment are capable of moving strings and substrings into memory from an input stream, within memory from one location to another, and from memory to an output stream. Patterns move matching substrings into memory using the assignment operator within the pattern definition. The assignment operator in a statement body assigns the (unmatched) source to the object. If the pattern match fails, or if no match is indicated, the entire source is assigned to the object.

When a source function is referenced as the sole element of a pattern match, and no assignment is indicated, the input pointer is advanced by the full length of a successful match. Otherwise, the reference is to a single record (up to, but not including, the LF), and the input pointer is advanced to the next record. If neither matched, nor assigned, an EOF test is done (success means that more source exists, and failure means that the EOF has been reached).

Beginner Encoding

See 11/24/01 (above). Every <sub> needs a list of beginner primitives that apply to it. In going recursive descent, a viable continuation can quickly be determined. The <sub> preface will indicate the list of primitives and the alternatives to which each applies. Only those alternatives will be attempted. The purpose of MATCH has got to be to generate a parse tree for any source input, each node of which is coded as follows:

```
[alt][len][@source] – results of a pattern reference (implicit alt)
[arb][arb][Token:K] – results of a primitive (primitive sets content)
[0FF][0FF][@object] – repeat construct delimits tokens emitted
```

In the normal course of recursive descent, a <sub> has a set of alternatives. Each alternative has a list of ([primitive], literal), or (primitive), that can begin it. The source is matched to this sequence, and the successful match indicates which alternatives of that <sub> should be tried. The list of tokens to be matched is determined at the outset, and each <sub> is only attempted if it is on a viable path.

The pattern data structure (after CANONIZE) is as follows:

```
       Begin:             [@Pattern][@WrapupData](4*)(12*)[@Subpattern]..
       Pattern:           [@NextAlt][suc][suc][@Alternative]..
                                  ... [ @00 ]
       Subpattern:        [@nextAlt][suc][suc][@Alternative]..
                                  ... [ @00 ]
       Alternative:       [alt#][suc].. [ 8 ]
                                  ...
                          [alt#][suc].. [ 0 ] <WrapupString>
       WrapupData:        [@WrapupSubDef].. <wrapup sub defs>
       where,
                 suc =    00      End of Alternative (end of pattern)
                          01-07   Invoke EXT0 (hide next 1-7 bytes)
                          08      End of Alternative (continue pattern)
                          09      Alt# Override (hide next 1 byte)
                          0A-0F   Invoke EXT1 (hide next 2-7 bytes)
                          10-1F   Token Override (set/save token pointer)
                          20      open option/repeat (close option if open)
                          21-60   literal byte
                          61-7A   call a primitive (26 allowed)
                          7B-7E   literal byte
                          7F      close a repeat (if open, else NOP)
                          80-83   Built-in primitives (REMATCH..EOS)
                          84-FF   match a subpattern (124 allowed)
          The first byte in an alt always gives the alt #.  However, before
             CANNONIZE, the alt # is positional unless an Alt # override is
             present.  An override = FF indicates an |= (same as prev alt #).
       A Token Override equal to the token # saves the token pointer.
       A Token Override elsewhere sets the token pointer.
          The Wrapup Data Structure is concatenated to the Pattern Data
             Structure at @WrapupData, and is defined similar to current PTRAN
             (except the initial entry address is omitted, see WRAP Update).
           4* = [@SRCFCNS][@OBJFCNS]
          12* = [@EXT0][@EXT1][@REMATCH][@CONTINUE][@EOL][@EOS]
```

The initial two sucs in a pattern vector are in the range 21-7E, indicating a literal byte or a primitive. The second suc may = 0 if the first is a literal, or if the first is a primitive and no literal directly follows the primitive. A match begins with a scan of the pattern vectors and an attempt to match an initial suc [suc]. Success vectors to the first alternative listed, and sequences through the alternatives until one is matched, or until all have been tried. Failure to match an initial suc [suc] is an immediate match failure.

Alternatives are listed in the order given in their definitions, however, if one alternative is the proper head of another, that alternative is made to follow the other (this forces a match to try the longer alternative first). When an entire definition contains no option or repeat constructs, nor any recursive definitions, it is considered a “fixed” grammar. Otherwise, it is a “variable” grammar. Overrides will automatically be inserted into fixed grammars so that alternatives of root patterns that have the same subpatterns in a different sequence, will conform to a canonical sequence (given by the first alternative defined for that root pattern).

A canonical sequence is defined by the top-level pattern definition. If it has two or more alternatives listing the same subpatterns in a different sequence, the second and subsequent alternatives will have overrides inserted to set the token pointer so that each alternative will emit the equivalent tokens into the same “slot” positions. If different alts of a subpattern have different length token sequences, the maximum number of slots will be reserved. The token slot following the match of a literal is always set to zero. This allows a “missing” primitive to be defined with a zero token. A slot otherwise skipped is undefined.

The ’ suffix may be used to indicate a repeated <sub> that is to rematch the same string it previously matched. The 080 code represents the ’ to CANONIZE, and CANONIZE automatically inserts the appropriate override code ahead of the initial subpattern reference and deletes the second reference (or reports an error for incorrect usage).

MATCH Update

Without changing a lot of the logic in XPL and DEXPL, I'd like to change MAIN significantly. PTRAN and OTRAN will be replaced by UPGEN (Universal Parser Generator). This code will contribute a MAIN that does what the current MAIN does, with only minor changes. MAIN will still be suitable for direct inclusion in both DEXPL and XPL. In addition, a module called CANONIZE will be suitable for inclusion in DEXPL. It will take a raw pattern/object definition, put it into canonical form, and turn it into an .OBJ file.

UPGEN source will be a combination of current PTRAN and OTRAN. A source program will begin with a preamble. This will be followed by statement definitions, then continue with pattern definitions like DEXPL, and finally with object definitions.

An initial object definition is the final part of each statement definition, however object subpatterns are defined after all statement definitions are complete. Three disjoint sets of labels are defined: The first set comes from the preamble: These labels are the GLOBAL symbols of the final .OBJ module. The form of a preamble statement is:

```
      <label> = <hex> | <binary> | ENTRY | SRCFCNS | OBJFCNS |
                          EXT0 | EXT1 | REMATCH | CONTINUE | EOL | EOS
```

The second set of labels are those that label patterns. These are referenced from statement definitions. The third set of labels are those that label object definitions. All of the labels in the second and third sets are local to the .OBJ file. The set derrived from the preamble is turned into a set of GLOBAL definitions that can be accessed by the other .OBJ files LINKed to the pattern object file. The first two alternatives, above, allow global symbols to be defined as given constants. The next alternative allows the main entry point (Begin:) of the pattern data structure to be given a name by which it can be referenced.

The two labels, SRCFCNS and OBJFCNS, refer to address vectors that must be contained in another .OBJ of the program. The addresses of these vectors are contained in the (4) bytes reserved at the beginning of the Pattern Definition. The addresses of the six built-in primitives follow in the next (12) bytes. This allows a set of Match and Wrapup functions to be tied to a given Pattern Definition (in case there are more than one in a single program).

There are six standard primitives (32 primitives in all). The external addresses of the standard primitives are contained in the Subpattern Vector (along with addresses for each of the subpatterns and the address of the Wrapup Data Structure). The preamble allows these functions to take on different names. Special syntax in a statement definition allows these primitives to be referenced. The pattern element "," indicates CONTINUE, a ";" indicates EOL, and a "." indicates EOS. The only way the functions EXT0 and EXT1 can be invoked is to use the 'hex' construct and force a string of byte codes into the object pattern being defined. Any use of 'hex' runs the risk that a pattern definition is corrupt, so it must be done with care, or with a special tool (such as DEXPL) .

MAIN (containing the MATCH engine) refers to certain external symbols that must be defined in the program to which it is LINKed. Likewise, during a MATCH and WRAPUP, certain register conventions must be followed by any primitive function invoked. The code and stack segments must be the same. BP+BC must indicate a pattern data structure, and BP+DE must indicate the initial pattern or subpattern to be matched. The first 16 bytes of $BP must be reserved for MATCH. SI,DS must indicate the next source byte. DI, ES must indicate the next object byte. AA and HL are scratch registers (except for a return value that may be contained in AA). When EQ is returned to MATCH, it indicates success and that no token is to be emitted. SI is positioned at the next source byte to be matched. LT indicates match failure. GT indicates that a token is to be emitted (SI = match continuation, token emitted is [HL][AA]). Since MATCH may be re-entered, all of its volatile information must be PUSHed and POPed if it is called recursively. This includes BC, DE, SI, DI, DS, ES, and certain $BP locations, however some of these are expected to remain constant over a normal recursion: BC, DS, and ES, for example. The first, third, and fifth address positions within $BP are reserved for copies of each of these. [Note: The details in this ¶ are tenative.]

NOTES: Discourage the use of leading null-matching primitives in pattern definitions. Delete the "block" header in pattern definitions. Encourage the use of the built-in extended primitives, and discourage the use of any primitive that swallows another. In the Wrapup structure, swallowing an EMIT is OK. Any other extension should use a second primitive. This means XPL has to convert its n and u functions to the built-in extended primitives.

WRAP Update

The initial handle on all wrapups is the wrapup string immediately following the final alternative of a pattern. The wrapup string is decoded as follows:

`00`		`. end of definition`
`01`		`; end of conditional`
`02`		`< action (push a pointer to the object code)`
`03`		`- action (decrement object pointer)`
`04`		`> action (pop a pointer to the object code)`
`05`		`+ action (bump the token source pointer)`
`06`		`EMIT() a 0-byte`
`07-0C`	`(1-6)`	`EMIT(1-6 bytes)`
`0D`	`(1)`	`XOR(byte) with last object byte`
`0E`	`(1)`	`GOTO byte (via wrapup address vector)`
`0F`	`(1)`	`CALL byte (via wrapup address vector)`
`10-1E`	`(1)`	`$n=byte (absolute conditional, n=0-E)`
`1F`	`(1)`	`invoke ? primitive with 1 parameter byte`
`20-2E`	`(2)`	`$n=byte,byte (absolute OR conditional, n=0-E)`
`2F`	`(2)`	`invoke ? primitive with 2 parameter bytes`
`30-3E`		`$<(0-E), relative conditional`
`3F`	`(1)`	`$<(0-FF), relative conditionl`
`40-4E`		`$=(0-E), relative conditional`
`4F`	`(1)`	`$=(0-FF), relative conditionl`
`50-5E`		`$>(0-E), relative conditional`
`5F`	`(1)`	`$>(0-FF), relative conditionl`
`60-7F`		`invoke primitive (`` ` `` a-z { | } ~ ? → $OBJFCNS)`
`1xxxyyyy`		`XOR($alt,xxx) alt=0-E, left shifted xxx bits.`
`1xxx1111`		`XOR($,xxx) current alt, left shifted xxx bits.`

This recodes the actions of the current interpretation, with two exceptions: The initial byte in DEXPL-generated strings is a flag byte that serves to terminate the pattern definition, represent a flag to be inserted in the source line, and indicate a null wrapup when appropriate. This exception will be replaced by inserting the code 1F (to invoke the ? with a non-blank flag to be inserted). Thus, nothing special is provided in the general case for this rather special action in the specific case of XPL. The second exception involves the codes 1F and 2F, which are not being interpreted this way in the current implementation. However, since the $F absolute alternative is not defined, these two codes are currently inaccessible. Note that $0 can never reference a primitive token slot (only an "alt" value). Again, the 1F code should probably be used for emitting object that results from a primitive. It might be nice to have CANONIZE verify such a correspondence (between wrapup primitives and pattern primitives, and between XOR operations and "alt" tokens).

Note: The + actions & fact that relative conditionals are followed only by the immediate clause, but absolute conditionals continue past a clause marker, need to be carefully documented so as to be well understood by the user.

NOTES (Futures)

1. After bridge XPL, the following syntax needs to be deprecated (gotten rid of): Any ,B or ,W suffixes; the keyword AGAIN (use LOOP instead); references (in XPLDEF) to patterns "w.rd" and "o.prim." All types of branch instructions should have IF, DO, and LOOP forms. The IFs and LOOPs should be "next" types, and the DOs should be "exit" types.

2. When XPL is updated to a new MATCH engine, let pass=0 emit XPATCH placeholders for all headers, and pass=1+ additional (reference) entries as necessary. The two types of XPATCH entries would be:

```
        [obj][body][@Tdef:0]
        [obj][slot][@Tref:K]
```

All entries of the first type would be emitted during Pass=0, where there would be a 1-to-1 correspondence between entries and source lines with a code < 0A. Only the "body" field would be entered by Pass=0. During Pass=1+, the rest of the information would be entered or updated. The "body" entries would record "obj" emitted FOR the current entry, "slot" entries would record "obj" emitted BETWEEN entries. Overflow now has to be considered: When slot=0, "obj" is recorded in the @Tref:K field, and the normal "obj" field is zero. Overflow is not a problem in "body" entries, as a line of source is incapable of generating 255 bytes of actual OBJECT.

This method neatly separates the tasks of interleaving object code and fixups into an .OBJ file, and tracking points in the code that must be referenced from other points in the code, or by the final line numbering scheme.

Pass=1 would mark the "body" byte to flag a "next" type of statement, and it would mark the "slot" byte to flag "exit" forward references (which would have to be fixed up if a "next" statement showed up as the target).

Left Recursion

Change the MATCH algorithm to something like the following:

1. Recursive descent (inserting nodes for each <sub>) until next element is a left recursion or a terminal.

    Left Recursion: Match alts of <sub> that begin with a terminal or a <sub> not on the current path.

    Terminal: Match the alternative.

2. Finish matching the entire alternative underway.

3. After Success:

    Recursion not pending: Return normal success to caller.

    Recursion pending: Attempt another iteration of 1 (insert a node, and do another Left Recursion).

4. After Failure: Normal backup and retry.

When MATCH encounters a <sub> reference and builds a Token node to represent it, it can temporarily store the <sub> code in the node. Each <sub> will provoke a scan back through the recent tokens to check consecutive <sub> references to see if the current <sub> is in the sequence. If it is, a left recursion has been detected.

Example #1: $A \rightarrow A\ b\ |\ c$

Given: c c b b c need to get: c ((c b) b) c where, ( ) enclose alt=0, and c is alt=1.

1. emit an A, take 1st alt

    A^0

2. detect A, invoke recursion: take 2nd alt & match c.

    $A^0\ A^0$ c

3. returning S: to "invoke recursion" continue 1st alt, but fail to find b, so backup & try 2nd alt at step #1

    A^1 c

4. starting at "c b b c" again emit an A and take 1st alt (steps 1-3 above, but succeed with 3.

    [A^1 c] $A^0\ A^0$ c b

5. continue 1st alt at 2nd recursion (iterating step #3) until b fails again, leaving:

    [A^1 c] [A^0 [A^0 [A^1 c] b] b]

6. matching the final c leaves the following token sequence:

    [A^1 c] [A^0 [A^0 [A^1 c] b] b] [A^1 c]

Example #2: $A \rightarrow A\ b\ |\ b$ (CANONIZE detects that b is a proper head of A b)

Given: b b b b need to get: (((b) b) b) b

1. emit an A, take 1st alt

    A^0

2. detect A, invoke recursion: take 2nd alt & match b.

    A^0 [A^1 b]

3. continue 1st alt at recursion (iterating step #2) until b fails, leaving:

    A^0 [A^0 [A^0 [A^1 b] b] b] b

Example #3: $A \rightarrow b \mid b\ A$ (CANONIZE detects that b is a proper head of b A)

Given: b b b b need to get: b (b (b (b)))

1. emit an A, take 2nd alt (CANONIZE makes it the first encountered) & match b

   A^1 b

2. repeat step #1 until match b fails.

   A^1 b [A^1 b [A^1 b [A^1 b [A^1 ?]]]]

3. backup:

   A^1 b [A^1 b [A^1 b $\rightarrow$]]

3. try next alternative (the 1st as it happens), leaving:

   A^1 b [A^1 b [A^1 b [A^0 b]]]

Product Definition

The following products may be derived from this project:

1. $25 for a history of microprocessors & a "toy" assembler[1] (the 8080 with IN/OUT to stdin/stdout).
2. $50 for more discussion of microprocessors and a full version of XPL (refers to 1).
3. $100 for UPGEN and a full explanation of how to define new translators (refers to 1-2).
4. $200 for DEXPL and a full explanation of how to define new assemblers (refers to 1-2).
5. $500 for the UPGEN source (includes 3).
6. $650 for all of the above (adds 1, 2, & 4 to option 5, saves $125).
7. $800 for the XPL and DEXPL source (includes 1-3).
8. $1000 for all of the above (combines 5 & 7, saves $300).

Any sale of derivative software, software that incorporates a .OBJ file or rewritten source that comes from any of these packages, must involve a separate license. The user of any of these packages is only allowed to use them as tools to create entirely original programs, or programs for personal, or non-profit use. The separate license will be sold on the basis of a "good faith" estimate of the hours of development time saved, times $25 / hour. No royalties will be requested. This is a one-time charge, and the use of only substantially similar content in another product will not incur a second license. A description of all of the code incorporated into the product, the amount the license is being sold for, both parties signatures, and the nature of the agreement will constitute the license. Either party has the right to reject the agreement or negotiate its price. The user does not have to obtain the license until the final product is offered for sale (or until derivative code is actually incorporated into a contracted deliverable).

An alternative to a 1-time charge is a royalty based charge. Given that a product incorporates my components, the fraction of development time added vs. total development time (my contribution + value added), gives the fraction of the royalty that should be taken by the user. My fraction should be the remainder. The royalty should not be less than 1/4th of the final product sales price. For example, someone generates a cross assembler for the XYZ chip. The product sells for $100 retail. Royalties should be $25 per copy. My fraction of that should be no less than 99% (XPL represents more than 100 weeks of development time, and the time to incorporate a new instruction set into a new product, and test it, should be no more than 1 week). Ratios for other derivative products might vary, this example shows the extreme.

[1] Includes regular and CPU display .OBJ modules with interface description and examples, also incorporates IN, OUT, and HLT instructions that invoke the appropriate INT 021 calls. Title: "Drive your own computer! Learn how to program the microproccessor that started the revolution. For beginners, not Dummies—An introduction to the highest level assembly language of all time."

Terminology: Call MATCH, PARSE. Call TOKENS, P.TREE.

UPGEN

Universal Parser Generator

This program will contain PARSE & CANONIZE. PARSE will contain the current MATCH & WRAPUP equivalents. The current MAIN will be gutted of MATCH & WRAPUP. The remainder will be rewritten slightly to conform with the new requirements of UPGEN. The following are design notes as development progresses.

CANONIZE

This routine accepts the parameters:

SI,DS = A 64K (read only) work area containing the input data structure.

DI,ES = A 64K work area to be used as scratch by CANONIZE, and finally containing the output .OBJ file.

BP,CS=SS: A ( ) byte data area available for use by CANONIZE.

CANONIZE attempts to verify, correct, and augment the input data structure. It then converts it to an FB .OBJ format. The following are design notes as development progresses.

```
FB,[  8, 0],<name of module    >
2n,[ @00  ],<ENTRY>
1n,[1-8, 0],<SRCFCNS,OBJFCNS,EXT0,EXT1,REMATCH,CONTINUE,EOL,EOS>
0n,[ @hex ],<name of hex constant>
...[0]
SRCDEF:    @PATDEF
           [@StatementDef]..
           [@NEXT]
           [@0]
PATDEF:    @OBJDEF
           [@Subpattern]..
           [@0]
OBJDEF:    @SUBDEF
           [@WrapupDef]..
           [@0]
SUBDEF:    @FIN <statement defs>
NEXT:      <subpatterns> <wrapup defs>
FIN:       [@x]
```

The general form of the input data structure is indicated above. CANONIZE will use the initial symbol table to begin the .OBJ file. It also uses the token values 1-8 in fixups for certain pre-defined vector locations. If one of these tokens is missing, an @0 constant is substituted. CANONIZE may add to this symbol table (and delete all of its external symbols, if so) to define points in its object code that caused an error. The symbol @hhhh, is entered as an unreferenced external (type 1n), to indicate an error at relative address hhhh-0100, such that when LINK attempts to link the .OBJ, it will list all of these error addresses and produce a valid .COM file (since no references to these externals will actually be present).

The pattern data structure (after CANONIZE) is as follows:

```
Begin:            [@Root][@Wrapups](8*@00)[@Subpattern]..
Wrapups:          [@WrapupDef]..
                  <wrapup defs>
Alternative:      [alt#][suc].. [ 8 ]
  (main)          ...
                  [alt#][suc].. [ 0 ] <WrapupString>
                  ...
Alternative:      [alt#][suc].. [ 8 ]
  (subs)          ...
                  [alt#][suc].. [ 0 ]
                  ...
Subpattern:       [@nextAlt][suc][suc][@Alternative].. [@00]
                  NextAlt: ...  [@00]
                  ...
Root:             [@NextAlt][suc][suc][@Alternative].. [@00]
                  NextAlt: ...  [@00]
```

```
where,
            suc =   00      End of Alternative (end of pattern)
                    01-07   Invoke EXT0 (hide next 1-7 bytes)
                    08      End of Alternative (continue pattern)
                    09      Alt# Override (hide next 1 byte)
                    0A-0F   Invoke EXT1 (hide next 2-7 bytes)
                    10-1F   Token Override (set token pointer)
                    20      open option/repeat (close option if open)
                    21-60   literal byte
                    61-7A   call a primitive (26 allowed)
                    7B-7E   literal byte
                    7F      close a repeat (if open, else NOP)
                    80-83   Built-in primitives (REMATCH..EOS)
                    84-FF   match a subpattern (124 allowed)
   The first byte in an alt always gives the alt #.  However, before
      CANNONIZE, the alt # is positional unless an Alt # override is
      present as the first element of the Alt.  Elsewhere, an EXT1 is
      indicated.  An override = FF indicates an |= (same as prev alt
      #).
A Token Override sets the token pointer.
   The Wrapup Data Structure is concatenated to the Pattern Data
      Structure at @WrapupData, and is defined similar to current PTRAN
      (except the initial entry address is omitted, see WRAP Update).
   8*@00 = [@SRCFCNS][@OBJFCNS]
      [@EXT0][@EXT1][@REMATCH][@CONTINUE][@EOL][@EOS]
```

The wrapup string is decoded as follows:

```
00                  . end of definition
01                  ; end of conditional
02                  < action (push a pointer to the object code)
03                  - action (decrement object pointer)
04                  > action (pop a pointer to the object code)
05                  + action (bump the token source pointer)
06                  EMIT() a 0-byte
07-0C      (1-6)    EMIT(1-6 bytes)
0D         (1)      XOR(byte) with last object byte
0E         (1)      GOTO byte (via wrapup address vector)
0F         (1)      CALL byte (via wrapup address vector)
10-1E      (1)      $n=byte (absolute conditional, n=0-E)
1F         (1)      invoke ? primitive with 1 parameter byte
20-2E      (2)      $n=byte,byte (absolute OR conditional, n=0-E)
2F         (2)      invoke ? primitive with 2 parameter bytes
30-3E               $<(0-E), relative conditional
3F         (1)      $<(0-FF), relative conditionl
40-4E               $=(0-E), relative conditional
4F         (1)      $=(0-FF), relative conditionl
50-5E               $>(0-E), relative conditional
5F         (1)      $>(0-FF), relative conditionl
60-7F               invoke primitive (` a-z { | } ~ ? → $OBJFCNS)
1xxxyyyy            XOR($alt,xxx) alt=0-E, left shifted xxx bits.
1xxx1111            XOR($,xxx) current alt, left shifted xxx bits.
```

Canonize Design

The source datastructure to Canonize should not exceed 48K with a maximum of 1-2K being the initial symbol table. Normally it will be much smaller than this, because the resulting .OBJ must be linked into a code module that is expected to execute in a 64K address space, including its stack. Although a few checks are made for corrupt data input, the assumption is that a valid and limited data structure is present.

The value @FIN determines how Canonize treats the data structure. If zero, the structure is assumed to be a Normal Grammar. Otherwise, a Fixed Grammar is “reshaped” and “verified.”

Once the components of the input datastructure have been moved into the work area, a routine called BUILD.EM is called to generate pattern headers for the entire set of subpatterns and statement definitions. Following this, any reference to the root, or a subpattern, is via a pattern header. Pattern headers contain references to individual alternatives of pattern definitions. The sequence of references within a header clause reflects any "proper head" relationships that exist. Left recursion always has some other alternative as a "proper head." If not, the grammar has an error. An alternative that **has** a "proper head" is always referenced **before** its "proper head."

BUILD.EM performs a complete recursive descent traversal from each subpattern and then the root definition. When a definition is encountered, it is either new, converted, or in the process of being converted. Definitions whose alternatives all begin with literals and primitives (no subpatterns), can be completely converted when first encountered. When an alternative that begins with a subpattern reference is encountered, a recursion takes place. If the reference is to an unconverted definition, its conversion is begun. Otherwise, there are two cases: The definition has been completely converted, in which case its beginners can all be returned, or its conversion is in progress, in which case a left recursion exists. A conversion in progress is flagged with a "placeholder."

When a conversion is begun, each alternative that begins with a subpattern reference is automatically marked with a placeholder (an @Index to the next header clause, an @00, and an @Alternative to the referenced alternative of the subpattern). Any alternatives that do not require a placeholder are handled first. Then, recursion is taken to handle each placeholder in turn. When the traversal encounters a placeholder, it is the case that all beginners to that point have been returned. Thus, the traversal can return from the definition at that level avoiding any recursive loop.

The purpose of the header structure is to indicate the possible alternatives that should be matched, given an initial literal or primitive. Thus, all possible beginners of a pattern are determined, and each one is used to define a clause that lists all the alternatives that could possibly match, given that beginner. A beginner (stored as [suc][suc] in the header structure) is the first literal byte or a primitive followed by a literal or primitive if a second terminal is defined. Thus, only if the first terminal in a definition is a primitive will the second [suc] be non-zero. While the first [suc] = 0, the clause is considered to be a placeholder.

The intermediate data structure (after MOVE.MORE) has a vector of statement definition addresses and another vector of subpattern definition addresses. The subpattern vector is sequenced through first, then the initial header definition is begun. After subpattern conversion, the index of the "next available" address is stored into $0 of the data structure. This becomes the index of the root definition. As header clauses are defined and extended, word values are inserted and the entire header area above that point is moved up two bytes (necessitating that all pointers above that point be increased by a count of two). By handling the statement definition vector last, this design is not too computationally expensive, and it is fairly simple conceptually.

Design Updates (to supersede anything said above)

In patterns, the REMATCH built-in normally corresponds to a fixed grammar. The sequence <subpattern> … <subpattern> <080> will be replaced by (first <subpattern> must be the same as the second, else error) the sequence <subpattern> <080> … <080>, where the second <080> must not be preceded by any <subpattern> (it will normally be preceded by a literal). The design change is that a Token Override is an unconditional override, not a command to "Save Token Pointer" when it happens to equal the current Token Number.

Only the pattern elements 21-7E may be statement beginners. The codes 1-7, 9-F, and 80-83 are disallowed as statement beginners. If the traversal for beginners returns any disallowed code, it simply continues until a valid code is returned, or until 00 or 08 (end of alternative) is returned (in which case the beginner may only be a single code in length). An empty alternative (except a final alternative) is deleted from the grammar. An alternative with no beginner will cause an error to be reported (including an empty final alternative).

● ● ● ● ● ● ● ● ● ● ● ● ●

(06/15/15) Much of the nearby text above may not apply, and will be deleted in the future. Text not deleted might be edited in the future. A **DEXPL** archive will be created when the (XPMode) work area is complete and self-making.

Current To-Do List

1. Get all components self-making again (in XP Mode).

2. Update design and fix bugs. Get that self-making in XP Mode.

3. Develop a cross-compiler that runs on the current Windows 7+.

4. Get the updated components self-making in the W7P Mode.

Review & Update of MAKE Process

The immediate objective is to get **MAKE** and **MAKE$** working (in XPMode) so that all the code modules are self-making. Spent the day reading the old log.txt. Tomorrow, I'll work on the init, make, and other .bat files.

(06/16/15) Got working directory integrity on XPMode (see Gary's Log). I'll experiment with it and **UPLOAD** or **DOWNLOAD** as I need to archive good results or restore if things become badly messed up. Eventually, I'll develop an Archive method to track development. For now, **UPLOAD** produces the archive. And disk backups produce the archives for this file and the latest fileset on the D: drive.

The original file set is at: D:\XPL\1972-02\XPL (+subdirectories).

Had to **DOWNLOAD** a couple of times due errors in **MAKE$**. 1st error was an instruction in **MAIN.SRC** (RRC instead of RR`). I need to perform the entire **MAKE** in steps. Found BR86.DEF in DEXPL/XPL. It's the same as the root.

(06/17/15) The sub directory structure within DEXL is: COMMON, DEXPL (PTRAN, OTRAN), PRINT, LINK, XREF, and XPL. The root MAKE visits each directory in endorder sequence (DEXPL after its subdirectories). Work in the PRINT and XREF subdirectories and on their components will be deferred until (possibly much) later.

The difference between MAKE and MAKE$ is that MAKE should execute from …\XPL\COM, and MAKE$ should execute ncw components left in a subdirectory by a previous MAKE. Compares of components should be between the old (proven to work) and new (moved to a local subdirectory). I shall proceed to edit all the MAKE files until I'm convinced that this is what they do. Remember, invoked files (.COM and .BAT) come from the local directory, or an explicit directory relative to the local directory, or from the current PATH (in that search order).

COMMON

For the working history, see the LOG in the directory. This directory contains source to clone a startup module (template code for ASCII Output to the console) called MAIN. The .COM produced is not particularly useful. It demonstrates compiling and linking the true common components: LF, SF, and FF. The COM subdirectory will contain **MAIN.OBJ** and **MAIN.COM** after any MAKE. These files can be compared to the files generated in the root directory after a 2nd MAKE. After changing the MAKEs, **COMP MAIN.COM COM** shows that the files are identical.

PTRAN

Reviewed LOG without much enlightenment, but it's there as a possible resource. Moved **DEL TMP** to beginning of Make files. Deleted **RESET.BAT** and **UPDATE.BAT**. Changed Make files to move all .com and .obj files to .\COM before creating new .com and .obj files. In particular, this makes the previous **PTRAN.COM** non-local, and that will be the one executed by MAKE$. A local flush.bat needs to delete the local copies to prevent them from redefining the copies in .COM after any MAKE that is not intended to be kept.

Note: subdirectories .OBJ, .COM, and OLD.TXT had a number of files. If anythng seems missing (development before or after the current state), re-visit the originals of these directories. Here and now, these subdirectories are deleted (.COM is now populated from the latest MAKE). Browse the originals someday out of curiosity.

OTRAN

Cleaned out the directory. Same Notes for the deleted files as above. Made changes to **MAKE.BAT**, and got **LINK** errors @0960 and @09EF. Obviously, the result did not compare with the working version, and is unacceptable.

OTRAN is built from its own entry template, the same **MAIN** and **PREP** as **PTRAN**, the common **LF**, **SF**, and **FF** objects, and its own **BODY**, **BACK**, **MORE**, and **USAGE**. **LINK** should see the following list of .OBJ files: **OTRAN** (**LF FF SF MAIN** (**MORE** (**PREP BODY BACK USAGE**))). What I need to do is compare the DEBUG binary of **OTRAN.COM** in its local directory and in the top .COM directory, and try to suss out where the code modules actually are, and why an overlap occurred.

Repaired OTRAN.SRC by ordering link modules: LF, SF, FF, MAIN. The .COM produced does not match the old top .COM, but it produces identical resuts given identical input .OT files, and appears to work OK. Once the entire program set is self-making, I should update:

D:\XPL\COM\OTRAN.COM with D:\XPL\DEXPL\DEXPL\OTRAN.COM.

However, both the local Makes do appear to work OK.

DEXPL\DEXPL

Note: this is the root directory for developing **DEXPL.COM**. The next higher **DEXPL** directory contains subdirectories for developing **XPL.COM** and **LINK.COM** (and **COMMON**, **PRINT**, and **XREF**). I'll visit them later. First, this directory needs a clean Make. Then it needs a general cleanup.

(06/18/15) After a clean **MAKE** and **MAKE$**, all files not involved were deleted. There were several old .txt files that could contain useful information. If such information is sought in the future, it should be accessed from (find) the "original file set" (above).

LINK

Replaced an old RRC operator with the RR` operator to get a clean Makes. The directory was also cleaned.

XPL

Updated Make files to get clean Makes. The directory was also cleaned.

PRINT

Updated source language conventions. Updated Make files to get clean Makes. Cleaned the directory.

XREF

Updated source language conventions. Updated Make files to get clean Makes. Cleaned the directory.

DEXPL

Makes in this directory simply call the makes in all of its subdirectories. The **PRINT** and **XREF** programs need further development. All Makes now work, and **XPL** is self-building.

Backups and Archiving

(06/19/15) ARC = 15-06-18X is the fully populated starting point for this leg of XPL development. It should contain a set of self-making files, and all of the debris left over from the leg abandoned in 2002.

(06/19/15) Today, the **.BAT** files **UPLOAD**, **DOWNLOAD**, **ARC**, **DARC**, and **XARC** (contained in the root DEXPL) need to be perfected. Note the current working directory is XPMode…\XPL\DEXPL.

UPLOAD this script completely replaces the D:\XPL\BAT, \COM, and \DEXPL directories with their current XPMode equivalents.

DOWNLOAD removes BAT, COM, and all the subdirectories of DEXPL and does a /D/S copy of them from W7Pro to the local XPMode counterparts.

ARC saves a small (2nd parameter = A, B, …) or large (2nd parameter missing) collection of files and subdirectories of DEXPL to the current D:\XPL\ARCHIVE.

DARC displays the folder contents of the W7Pro D:\XPL\ARCHIVE.

XARC extracts all files (no 2nd parameter) from a named (1st parameter) archive in the W7Pro D:\XPL\ARCHIVE. A 2nd parameter gets a single file from within the named archive, and a 3rd parameter renames the file copied.

The End

● ☐ ☑ ✓ @ ¶ ∃ × Δ Σ ∞ ½ ⅓ ⅔ ¼ ¾ ⅛ ⅜ ⅝ ⅞ º π