PL/65

USER'S MANUAL

PL/65

Rockwell International

$10.00
Document No. 29650 N37
November 1978

TABLE OF CONTENTS

INDEX

SECTION 1

INTRODUCTION

A high level systems implementation language, PL/65, has been developed for use on the Rockwell SYSTEM 65 Microcomputer Development System. This document describes an implementation of that language for the R6500 family of microprocessors. The language resembles PL/1 and Algol in general form, but a large number of simplifying assumptions have been made to avoid unnecessary complexity and to ease compiler implementation. The result is a mid-level language which has the power and flexibility of Assembler and the structuring potential of a high-level language.

All language features are aimed at improving the productivity of the systems programmer by simplifying the writing of systems normally written in Assembler. PL/65 compilers produce assembler code rather than object code. Thus, symbol table manipulation in the compiler is avoided. Furthermore, this allows the systems programmer to enhance or debug at the assembler language level if necessary. In fact the programmer may revert to assembler language whenever it is expedient to do so.

In spite of the simplicity of the language, the systems programmer is able to structure programs for readability and logical organization without sacrificing run-time efficiency. Program logic essentially becomes self-documenting. Control structures for conditional and iterative looping and the IF-THEN-ELSE conditional coupled with a simplified block structure add greatly to the language potential for highly structured programs. This document describes the language features of PL/65. Consult Section 4 for PL/65 installation and operating instructions on SYSTEM 65.

This PL/65 Language Reference Manual assumes user familiarity with programming in a high-level language.

SECTION 2

STRUCTURE OF PL/65 PROGRAMS

A PL/65 program is a collection of PL/65 statements. The compiler does not impose an ordering of the program, hence the systems programmer bears the responsibility for correctly formed programs. The general structure of PL/65 programs is illustrated in the following ascending order bubble sort routine.

```
PAGE '**SORT**';
;
"ASCENDING ORDER SORT";
;
DECLARE F, I, TMP;
ENTRY $200;
;
N=N-2; "SET TERMINAL VALUE FOR LOOP";
F = 1; "SET FLAG";
WHILE F = 1
          DO;
          F = 0;
                    FOR I = 0 TO N
                    BEGIN;
                    IF B[I] > B[I+1] THEN
                              BEGIN;
                              F = 1;
                              TMP = B[I];
                              B[I]=B[I+1];
                              B[I+1]=TMP;
                              END;
                    END;
         END;
N: DATA 10; "N IS NUMBER OF SORT ELEMENTS";
B: DATA 23,55,36,28,54,39,99,86,21,67;
;
EXIT;
```

Example 1. ASCENDING ORDER BUBBLE SORT

Programs normally start with an ENTRY statement; and terminate with an EXIT; statement. Their use is explained in later sections. The sample sort routine above shows program structures common to many high

level languages. Particular structures shown are: assignment, integer arithmetic, conditional looping (WHILE-DO), iterative looping (FOR), conditional execution (IF-THEN), collective execution (BEGIN-END), linear array manipulation, data area declaration (DECLARE), and array initialization (DATA). Statements are separated by semi-colons and optional comments. Comments are delimited by double quotes and terminated by semi-colons. The compiler is not column sensitive so statements can be indented for readability. TABS can also be used. The following routine is designed for execution on an R6500. A transistor switch and relay are assumed to be connected to pin 1 of an R6522 Port B Data Register. The program cycles through an array T for a switch position (1 = on, 0 = off) and a time delay (in seconds).

```
PAGE 'TOGGLE SWITCH';
;
DEF DRB=$A000, DDRB = $A002
DEF T1CL=$A004, T1CH = $A005
DEF ACR=$A00B, PCR = $A00C
DEF IFR=$A00D
DEFINE *=$10;
DCL J,K,L;
DCL TIME;
ENTRY;
       UDDRB=$FF;
ST: CLEAR ,CARRY;
       T1CL=$50
       T1CH=$C3
       FOR J=0 TO 10 BY 2
              BEGIN;
              DRB=T[J];
              TIME=T[J+1];
              CALL DELAY;
              END;
GO TO ST;
;
DELAY: FOR K=1 TO TIME
       BEGIN;
              "FOLLOWING IS A 1 SECOND DELAY";
              FOR L=1 TO 20
              BEGIN;
              WHILE BIT [6] OF IFR ^=0;
       END;
RETURN;
;
"FOLLOWING IS SWITCH POSITION AND TIME DELAY";
DATA 1,5,0,10,1,5,0,15,1,5,0,2;
EXIT;
```

Example 2. SWITCHING WITH TIME DELAY

Assembly language generated by the compiler for the above routines is given in Appendix D. These routines were selected specifically to show some of the commonality between PL/65 and other languages. The following routine illustrates some of the features which make PL/65 unique.

```
PAGE '*MONITOR SEGMENT';
;
"SEGMENT OF A MONITOR PROGRAM";
RST:  .X=$FF;
        .S=.X;
        SPUSER=.X;
        CALL INITS;
DETCPS:  CNTH30=$FF;
        .A=$01;
DET1: IFF BIT[SAD] THEN START;
        IFF .N THEN DET1;
        .A=$FC;
DET3: CLEAR .CARRY;
        .A+$01;
        IFF ^.C THEN DET2;
        CNTH30+1;
DET2: .Y=SAD;
        IFF ^.N THEN DET3;
        CNTL30=.A;
        .X=$08;
        CALL GET5;
```

Example 3. SEGMENT OF A MONITOR PROGRAM

This example shows some of the extended features of PL/65 using named bits and named registers. Also shown is a special form of the IF-THEN conditional statement. These features are explained in detail in later sections.

SECTION 3

STATEMENT TYPES IN PL/65

Statements in PL/65 can be grouped into seven classes:

- Declaration
  - DECLARE
  - DEFINE
  - DATA
  - comment
- Assignment
  - byte move
  - multiple byte move
- Imperative
  - SHIFT
  - ROTATE
  - CLEAR
  - SET
  - CODE
  - HALT
  - WAIT
  - STACK
  - UNSTACK
  - INC
  - INCW
  - DEC
  - DECW
  - PULSE
- Specification
  - ENTRY
  - EXIT
- Conditional
  - IF-THEN-ELSE
  - IFF-THEN
  - CASE
- Branching
  - GOTO
  - CALL
  - RETURN
  - RTI
  - BREAK

- Looping
  FOR-TO-BY
  WHILE

Sections 3.1 through 3.7 provide a very general description of PL/65 statement types and their function. A more formal definition and detailed examples are given in Appendix A.

In the descriptions to follow, VARIABLE refers to either a name (label) or an integer. An integer may be decimal, hexadecimal (number preceded with a '$'), or binary (number preceded with a '%'). EXPRESSION means variables connected with arithmetic or logical operators. In an expression, the first term may normally be subscripted and sometimes may be indirect if that mode is not explicitly restricted. SUBSCRIPT refers to a variable or expression (which is non-indirect and non-subscripted) which reduces to a one byte value. STATEMENT is a single PL/65 statement and BLOCK refers to a group of PL/65 statements delimited with the words DO;-END; or BEGIN;-END;.

3.1 DECLARATIONS

The only data types in PL/65 are bytes and linear arrays of bytes. The compiler does not verify data type and there are no implicit type conversions. The sole purposes of the DECLARE command are to reserve and initialize data storage. The statement DECLARE ALPHA; reserves a byte (8 bits) of storage which can be referenced symbolically by ALPHA. The statement DECLARE GAMMA BYTE INITIAL[13]; reserves a byte (8 bits) and initializes that byte to decimal 13. Areas for byte arrays and character arrays can be reserved and initialized. Sample array declaration and character array initialization are shown by the following:

```
    DECLARE IOTA[40];
DCL R CHARACTER['RAIN'];
```

In the above example IOTA is an array of 40 bytes and there is no initialization. The symbol R references the first character of the string 'RAIN'. Pairs of bytes (words) can also be reserved; DCL SUB WORD[30]; reserves 30 pairs of bytes. Words can also be initialized as shown by:

```
     DCL D WORD INIT[5];
("INIT" stands for "INITIAL".)
```

The DEFINE statement is equivalent to an assembly language equate. The statement DEFINE J=32; defines the symbol J to have the value decimal 32. Virtually no syntax checking is done by the compiler on the DEFINE expression; any string of symbols is accepted by the compiler without error. Errors in expression formation are noted at assembly time rather than at translate time. Note that the PL/65 compiler produces assembly code rather than machine code.

Arrays can be initialized with a DATA statement as in the example:

```
MT:  DATA 24, 36, 27, 54;
```

Here, a 4 byte array named MT is defined and initialized. All arrays are zero origin. Hence, MT[0] is 24 and MT[3] is 54. Except for the facility to initialize arrays, the DATA statement serves the same purpose as DECLARE. Word arrays can be initialized with the DATAW statement.

Comments in PL/65 are strings of characters delimited by double quotes as in "SET MAX VALUE TO 5";. Comments require terminating semicolons as do all other statement types. Comments may be freely used wherever statements can occur, but comments cannot be inserted between syntactic elements of statements. Spacing of the listing may be accomplished with lines that contain only a semicolon (null statement).

3.2 ASSIGNMENT

The data movement instruction in PL/65 is the assignment statement which has been enhanced to make data movement more convenient. The statement B = C; means move the byte at location C to location B. Also, B.1 = C; means move the byte at location C to location B, and B.3 = C; means move 3 consecutive bytes starting at location C. The more general form B.K = C; specifies movement of K bytes.

A variable name with decimal point is said to be "quantified" and the expression to the right of the decimal point is the quantifier. In the absence of a quantifier, single byte operations are implied. The quantifier can be a PL/65 arithmetic expression, or simply a variable name or constant.

An expression is an operand followed by an arbitrary number of operator-operand pairs. The operands can be variable names of integers. A typical expression in PL/65 is B + 3 - C which is evaluated as if it were (B + 3) - C. Similarly D - E - F is evaluated as (D - E) - F. Thus, the statement B.K - 3 + J = C; will cause (K-3)+J bytes to be moved from byte locations starting at C to consecutive locations starting at B.

> NOTE:
>
> Since the symbols 'A', 'X', 'Y', 'S', and 'P' have special meaning in R6500 Assembler they should not be used as variable names in PL/65.

The right side of an assignment can be a PL/65 expression, as in:

```
B = C + D - E + 24;
```

Expressions are evaluated left to right, in the order in which operators are encountered. Parentheses are allowed in assignment statements. The major restriction is that the right side of an assignment statement must not begin with a left parenthesis. Thus

```
B = (C + (D -5));
```

is invalid. Note that parenthetical groups lower code efficiency and should be used with appropriate care.

Byte arrays and words, character strings, and quantified variables can be used in arithmetic expressions. The operators are permitted

+, -, .AND [logical AND], .OR [logical OR], and .EOR or .XOR [exclusive OR].

Operands follow assembly language conventions: a prefix $ indicates a hexadecimal constant, % indicates binary, and # signifies that the address of the symbol is to be used rather than the value. As noted, arrays of bytes are always zero origin. The statement B[0] = C[0]; has the same effect as B = C; Further, B[0].1 = C; has the same effect as B.1 = C; However, byte assignments should not be so quantified or reduced code efficiency will result.

Subscripts can be integers, variable names, or expressions. Thus, the most general form of byte assignment is

```
B[expl].exp2 = C[exp3];  or
B[expl].exp2 = C operator D;
```

Note that only one subscripted variable is allowed on the right side of the '=' and, if present, it must be the only variable. The statement B[expl] = C[exp2] is a single byte assignment. Unlike code generated by some compilers there is no subscript range checking for arrays. Thus, an errant array reference could refer to executable code as data. Since only byte arithmetic is used, the largest value a subscript may have is 255.

Indirect single byte assignments are permitted with the operators '@' or '&'. '&' means that the variable is already in page zero. '@' means that the variable is not in page zero and will be moved to a temporary page zero location before being used. The most general form of indirect assignment is:

```
@Variable[sub] = @variable+[sub]
```

or

```
@variable[sub] = @variable+expression;
```

where either '@' may be replaced with '&' if appropriate. Subscripts

in indirect assignments will always generate code corresponding to the INDIRECT Y form in the Assembler. For example:

```
@B[5]
```

addresses the fifth byte after the location pointed to by B. If location B contains $1000 (stored as $00, $10), @B[5] refers to location $1005.

For word assignments, the alternate form '==' may be used for the '.2' quantifier. Thus:

```
WORD1.2 = WORD2;
WORD1 == WORD2;
```

are equivalent statements.

3.3 IMPERATIVE

Each computer model has its own unique instruction set and format. However, certain functions are common to a broad spectrum of machines. That set has been included in the PL/65 language as a convenience to the programmer. Bit manipulation is performed with the instructions SHIFT, ROTATE, CLEAR, and SET, which operate on single bytes. The instruction SHL B.3; is a shift left of 3 bit positions with zero fill. ROL B.2; is an end-around shift left 2 bit positions. The default quantifier is 1. Bits of a byte can be cleared or set by providing a mask for the variable name as in

```
SET BITS[203] OF B;
```

which will set bit positions corresponding to ones in the binary representation of the value 203 decimal (i.e., 11001011).

The CODE statement allows the user to specify assembler code directly. Any information in single quotes is copied directly to the output file without any processing. Hence, assembler code can be inserted in single quotes. The keyword CODE is optional. Blocks of assembler code may be passed between two lines which have an '*' in the first column.

The HALT; command is a termination of program execution. For the R6500 processor the code generated is a jump to self [JMP *].

A WAIT instruction is included in PL/65 to help handle asynchronous interrupts. A WAIT is a conditional delay for an asynchronous event. The form is WAIT ON BITS[100] OF T; In this case, T would normally be some register which can be affected by external events. As soon as any of the specified mask bits of T are '1' the wait is terminated and execution continues. At times a delay may be required until all of the bits specified are set. Also, it is sometimes necessary to wait until the bits are clear (i.e., 0). Four options with the WAIT statement are shown in the examples below. Note that default options are "ALL" and "SET".

```
WAIT ON ANY BITS[ALPHA] SET OF BETA;
WAIT ON ANY BITS[ALPHA] CLEAR OF BETA;
WAIT ON ALL BITS[ALPHA] SET OF BETA;
WAIT ON ALL BITS[ALPHA] CLEAR OF BETA;
```

Two stack manipulation instructions are present in PL/65 and they are extremely useful for saving and restoring register values on entry or exit from subroutines. Examples are:

```
STACK R0, R1, R2;
STACK WORD R1;
STACK R2, R1, R0;
UNSTACK R0,R1,R2 ;
UNSTACK WORD R1;
```

3.4 SPECIFICATION

Normally the first statement of a PL/65 program is ENTRY. Actions performed correspond to initialization functions. For the R6500 the ENTRY statement initializes the instruction counter to $0200 (page 2), clears decimal mode, and initializes the stack pointer to $FF. If programs are to be assembled at a location other than $0200 the address can be specified as for example: ENTRY $0300;. The EXIT statement generates an assembler ".END" instruction.

3.5 CONDITIONAL

The decision structure common to many high-level languages such as Algol and PL/1 is the IF-THEN-ELSE conditional. Here is one form of the conditional for single byte tests:

```
IF B<C THEN
D = D + 1;
ELSE
B= B - 1;
```

The general form is:

```
IF expl relopr exp2 THEN
Statement [or block]
ELSE
Statement [or block]
```

The ELSE clause is optional and may be omitted. The expressions expl and exp2 are as defined in Section 3.2 with the restriction that exp2 may not be indirect. The relopr (relational operator) must be one of the following:

```
<  less than
<= less than or equal to
>  greater than
>= greater than or equal to
=  equal to
^= not equal to
```

Multiple byte comparisons may be made on direct values only (i.e., indirection is not allowed) and the comparison must be chosen from:

```
=  equal to
^= not equal to
```

Conditionals can be nested to any depth as illustrated by:

```
IF P > B + C THEN
IF  C < 5 THEN
       IF E - 4  = F THEN
       P = E + F .EOR G;
```

Collective conditional execution is accomplished by using BEGIN-END blocks.

```
 IF N < MAX THEN
BEGIN;
SUM = SUM + P;
N = N + 1;
P = B[N];
END;
```

An alternate form of conditional execution based on a computed value is provided as a form of CASE statement. An example is:

```
CASE [N] [LB1, LB2, LB3];
```

In this example a branch is made to label LB1, LB2, or LB3, depending on whether N is 1, or 2. The value of the expression must reduce to an integer for which there is a label in the list. There is no range checking on the computed value and random execution errors will occur if the range is improper. Any number of labels is permitted and the conditional value can be an expression. The PL/65 CASE statement is analagous to a computed GO TO in FORTRAN; in fact, the keywords "GO TO" may be used in place of "CASE".

3.6 BRANCHING

Statements may be labeled with a name of 1 to 6 alphanumeric characters followed by a colon. Multiple labels are not permitted. An unconditional branch to a specific statement can be made with a GO TO, e.g., GO TO

START;. Indirect branches are supported, e.g., GOTO @START;. The unconditional branch should be avoided when possible since it often obscures the logic of the program.

Subroutine calls are supported with the commands CALL and RETURN.

```
 ...
CALL SUB1;
 ...
B1: ...
RETURN;
```

The CALL executes a Jump to Subroutine (JSR) and acts as an absolute branch to the specified label. The RETURN executes a Return from Subroutine (RTS) and causes execution to resume at the statement following the call. Since all variables are global to the main program and all subroutines, there is no need or provision for parameter passing. An RTI (return from interrupt) instruction is available, also.

3.7 LOOPING

Two forms of looping constructs are implemented in PL/65. An example of the iterative form is:

```
FOR I = 1 TO 13 BY 2
BEGIN;
C[I];
SUM = SUM + B;
END;
```

The general form is:

```
FOR name = exp1 TO variable BY variable
statement [or block]
```

where exp1 is the initial value for the variable name, exp2 is a test or terminal value and exp3 is the value of the variable incremented each iteration. As in PL/1, the test is made prior to first execution of the statement and hence there are cases where the statement may not be executed at all as in the example:

```
FOR J = 5 TO 4 ... ;
```

The BY clause is optional; an increment of 1 is assumed if it is omitted. FOR-LOOPS may be nested to any depth. The loop is executed until the value of the loop variable exceeds the value of the terminal expression. Thus,

```
 FOR J = 1 TO 1
...
```

is a non-terminating loop with exit from the loop by a direct GOTO statement presumably. The second form of looping has no iteration and is shown by:

```
WHILE B < C + D
DO ;
B = B + 1;
C = C - 2;
END;
```

The loop block is executed as long as the conditional expression is true. As in the FOR-LOOP the test is made prior to first execution and hence the block may be bypassed with no execution.

The general form is:

```
WHILE variable relopr variable
DO;
Statement[s]
END;
```

The relopr (relational operator) and variables are as defined earlier.

3.8 COMPILER GENERATED LABELS

Note that the compiler generates labels which are of the form 'ZZ' followed by a number. Therefore, it is strongly recommended that any labels generated by the programmer are of a different form.

3.9 PAGE ZERO UTILIZATION

PL/65 imposes certain minor restrictions on the use of page zero: The first is that locations $0 through $5 are used as temporary locations by the compiler. It is recommended that programs start their page zero locations at $10. This may be easily done with the DEF *=$10; statement. The second restriction applies to PL/65 on the SYSTEM 65, but is a good programming practice on any machine. Since the compiler uses page zero when it is running, the program must not generate values into page zero by using the INIT option of the DECLARE statement or any other statement which cause corruption of page zero. Thus, the general rule is that the program should initialize page zero when it is started, and not when it is compiled.

SECTION 4

INSTALLATION AND OPERATION

The PL/65 Compiler is provided in object code form on mini-floppy diskette for direct installation on SYSTEM 65. Also included on the diskette is the object code for a PL/65 Optimizer program and a PL/65 Test program. The files are identified as:

FILE NAME	FILE DESCRIPTION
PL65Vn	PL/65 Object Code
OPTVn	Optimizer Object Code
TEST	PL/65 Test Program

4.1 LOADING THE PL/65 COMPILER

1. Install the PL/65 Compiler diskette into one of the two SYSTEM 65 disk drives.

> CAUTION:
>
> The PL/65 Compiler object code occupies 14K bytes of RAM -- from address $0200 to $3700. Be sure to save any data required within this address range before loading the PL/65 compiler.

2. Type L after display of the Monitor prompt. SYSTEM 65 will respond with:

```
<L>IN=
```

3. Respond to the input prompts:

```
<L>IN= F  FILE= PL65Vn  DISK=1
```

SYSTEM 65 will load the PL/65 Compiler object code into RAM and will display the Monitor prompt at the end of the load (after approximately one minute):

```
<
```

4. The PL/65 Compiler disk may be removed from the SYSTEM 65 disk drive until subsequent load of the compiler or optimizer is required.

4.2 OPERATING THE PL/65 COMPILER

1. If the PL/65 application source program is contained on diskette, install the diskette in one of the SYSTEM 65 disk drives.

2. Enter PL/65 Compiler by typing 7. SYSTEM 65 will respond with:

```
<7>
PL/65 [VX]
IN=
```

> NOTE:
>
> X will be a version number.

3. Enter the code of the input device containing the PL/65 application source program. For example, if a source program with file name PLSRC is contained on a diskette and is mounted in SYSTEM 65 disk drive 1, type in this data in response to SYSTEM 65 prompts:

```
IN=F  FILE=PLSRC  DISK=1  OUT=
```

4. Enter the code of the output device where the formatted PL/65 source program listing is to be directed. For example, if the output is to be printed, type P. SYSTEM 65 will respond with:

```
IN= F  FILE=PLSRC  DISK= 1  OUT=P
```

SYSTEM 65 will proceed to list the formatted PL/65 source code.

5. At the completion of listing the formatted source program, PL/65 will display the error count in decimal and ask for an output device code.

```
ERRORS= NN   where NN= 00 TO 99
OUT=
```

6. Enter the code of the output device that the PL/65 compiler output (R6500 Assembler Source Statements) is to be directed. For example, if the output is to be directed to a diskette on SYSTEM 65 disk drive

2 with file name ALSRC, type in this data in response to SYSTEM 65 prompts:

```
OUT=F  FILE= ALSRC  DISK=2.
```

PL/65 will then compile the PL/65 application source program into R6500 assembly statements and at the end of the compilation, will display a disassembled instruction and the Monitor prompt:

```
=XXXX 4C 00 02  JMP $0200
<
```

If it is desired to optimize the assembler source code, follow the procedure described in Sections 4.3 and 4.4. Otherwise, the PL/65 compiler output assembler statements may be assembled in accordance with the SYSTEM 65 operating procedure (see Section 9 of the SYSTEM 65 User's Manual).

4.3 LOADING THE PL/65 OPTIMIZER

1. Ensure that the PL/65 Compiler diskette is installed into one of the two SYSTEM 65 disk drives.

2. Type L after display of the Monitor prompt:

```
<L>IN=
```

3. Respond to the input prompts:

```
<L>IN=F  FILE=OPTVn  DISK=[1,2]
```

SYSTEM 65 will load the PL/65 Optimizer object code into RAM and will display the Monitor prompt at the end of the load:

```
^
```

4. The PL/65 Compiler diskette may be removed from the SYSTEM 65 disk drive until subsequent load of the PL/65 Compiler or Optimizer object code is required.

4.4 OPERATING THE PL/65 OPTIMIZER

1. If the PL/65 compiler output assembler statements are contained on diskette, install the diskette in one of the SYSTEM 65 disk drives.

2. Enter the optimizer by typing 7. SYSTEM 65 will respond with:

```
<7>IN=
```

3. Enter the code of the input device containing the input assembler source program. For example, if the source program with a file name of ALSRC is contained on a diskette mounted in SYSTEM 65 disk drive 1, type in this data in response to SYSTEM 65 prompts:

```
<7>IN=F  FILE=ALSRC  DISK=1  OUT=
```

4. Enter the code of the output device which is to receive the optimized assembler source program. For example, if the output is to be stored on a diskette mounted in drive 1 with file name ALSRC', type in this data in response to SYSTEM 65 prompts:

```
<7>IN=F  FILE=ALSRC  DISK=1  OUT=F  FILE=ALSRC'  DISK=1
```

5. SYSTEM 65 will proceed to optimize the input assembler source program. At the end of the optimization process, SYSTEM 65 will respond with a decimal count of the number of instructions deleted:

```
COUNT = NN      Where NN = 00 TO 99.
=XXXX 4C 00 02 JMP 0200
```

The optimized assembly source statements may now be assembled in accordance with the SYSTEM 65 Assembler instructions (see Section 9 of the SYSTEM 65 User's Manual).

4.5 PL/65 TEST PROGRAM

The source code of the PL/65 Test Program may be used as a test file to become familiar with the PL/65 Compiler processing. Use this file as the PL/65 Compiler application source program in Section 4.2.

APPENDIX A

STATEMENT FORMATS

This section describes the format possibilities for the various types of statements. Examples are given to show most format types. The descriptions are alphabetized according to statement type. Brackets < > denote the field is optional and a slash / indicates that a choice is to be made from the alternatives given. Descriptions are in the following order:

A.1 ASSIGNMENT

The assignment statement is probably the most powerful and useful statement type in PL/65. This does not mean that it is also the hardest one to understand. There are certain restrictions on the types of assignment statements which are legal. These restrictions may, at first, seem to be arbitrary but are fairly easy to understand if you look at the code generated for a particular statement. In general, the restrictions are caused by the availability of only two index registers on the R6500 processor or the size of the index registers (8 bits). The examples below attempt to illustrate the types of statements which are legal but a few general rules may be stated:

1. The left term may always be subscripted. One subscripted term may appear on the right side. If present, it must be the first term and must be the only term in a multiple byte assignment. Subscripts must fall in the range 0 to 255.

2. Indirect operators are '@' which means the term is not located in page zero but will be moved to a page zero temporary location before being used. '&' means the term is already in page zero. This form results in greater code efficiency than does the first form.

3. Indirect terms may occur on either side of the statement and may be subscripted. They may only occur in single byte statements. As mentioned earlier, subscripted indirect statements generate code which uses the INDIRECT Y Mode. If an indirect term is used on the right side, it must be the first term.

4. Multiple byte statements may only have two terms on the right side. If a multiple byte expression is subscripted on the right side, only that term may appear.

5. The '==' form may be used to signify a two byte move. It generates exactly the same code that '.2=' would generate.

6. If a character string appears on the right side, it must be the only term.

7. Single byte assignments (the only form which allows more than two terms on the right side) may contain parenthetical groupings with the '(' and ')' operators subject to the restriction that '(' must always be preceded with a arithmetic or logical operator. This is the only place that parenthetical expressions are allowed.

8. The value of an address may be assigned with the '##' operator as shown below. A single byte value may be assigned with the '#' operator.

9. Note that parenthetical groupings and indirect terms on the right side lower code efficiency due to the necessity of saving intermediate results.

10. The abbreviated form:

    ALPHA+n; or ALPHA-n;

    may only be used on non-subscripted, non-indirect variables.

```
                    <[sub]> expression /
<label:> name               =array element/;
                    <.expr> 'characters'
```

Examples:

```
B = C; "SIMPLE BYTE TRANSFER";
@B = C; "BYTE TRANSFER INDIRECT DESTINATION";
B= @C; "INDIRECT SOURCE";
B= #C; "VALUE OF SYMBOL TO LOCATION B";
B = ##C; "ADDRESS ASSIGNMENT":
B= D + F; "SIMPLE ARITHMETIC AND ASSIGNMENT";
B= G[4]; "BYTE TRANSFER FROM ARRAY G ELEMENT 4";
B[J] = C; "TRANSFER TO ARRAY B ELEMENT J";
B.K = C; "TRANSFER OF K BYTES STARTING AT C";
B[J+5] = C; "MOVE BYTE AT C TO ARRAY B ELEMENT J+5";
B[J] = 'X'; "TRANSFER OF LITERAL BYTE 'X' TO B";
B[J+1].K-1 = 'ABCDEF'; "MOVEMENT OF K-1 LITERAL BYTES";
" LOGICAL OPERATORS ";
C = D .AND F;
C = D .OR F;
C = D .XOR F;
C = D .AND F .OR 2 .XOR $55;
B .AND 1;
B .OR 2;
B .EOR 2;
B+1;
B-1;
B+5;
B+J;
```

```
B-J;
B+J-K+5;
B.K = C-D; "ONLY TWO TERMS ON MULTI BYTE";
B.K = C[J]; "ONLY ONE TERM IF SUBSCRIPTED IN MULTI BYTE";
B[J].15 = C[J]; "ALSO LEGAL";
B.K = 5; "SET K BYTES ALL EQUAL TO 5";
@B = @C;
&B = &C;
&B = @C;
@B = &C;
@B[5] = @C[J];
&B[5] = &C[J];
@B[K] = C[5] +D -F;
&B[5] = C[5] +D -F;
&B = C[R]=D-6 .AND 3;
&B[G-3] = C[N]+D-6;
&B[3] = C[4] - (F-D+3);
B[5] = @C+D-(F-N+3);
B[5] = @C;
B[N] = &C;
```

A.2 BLOCK

A block is a collection of statements (one or more) to be treated as a single group.

```
                    BEGIN;/
<label:>                          statements END;
                    DO;
```

No distinction is made between keywords BEGIN and DO and either may be used. Nothing more than logical grouping is implied by blocks.

Examples:

```
FOR J = 1 TO 5
DO ;
B[J] = J;
B[J] = J-1;
END;
IF ALPHA = 0 THEN
BEGIN;
BB = $21; "NOTE HEX CONSTANT";
BC = 17;  "NOTE DECIMAL CONSTANT";
BC = %1011; "NOTE BINARY CONSTANT";
END;
```

A.3 CALL

Labeled blocks may be called as subroutines.

```
<label>:        CALL        name;
```

This statement is the same as GO TO except that when a call is made a return address is kept so that a RETURN command will cause execution to be returned to the statement following the call. Any labeled statement is acceptable as a target; however, a block is recommended for clarity.

Examples:

```
    CALL SUB1;
    CALL ALPHA;
    ...
B1: BEGIN;
    ...
    RETURN;
    END;
    PPHA: BEGIN;
    ...
    RETURN;
    END;
```

A.4 CASE

Selection from a number of branch targets based on a computed value is made possible by the CASE statement.

```
                CASE/
<label:>                    [expr] [label,label,...];
                GO TO
```

The name must represent a value corresponding to a position of a label in the label list. The first position is zero. Note that the blank before the '[' is required.

Examples:

```
CASE [N][L1,L2,L3]; "N MUST HAVE THE VALUE 0,1, OR 2";
GO TO [N][L1,L2,L3]; "ALTERNATIVE CODING FOR ABOVE";
GO TO [B][ALPHA,BETA]; "B MUST REPRESENT 0 OR 1";
GO TO [N+B-1][ALPHA,BETA,L1,L2,L3];
CASE [B[J+C .OR 40] +N .AND $3F] [L1,L2,L3];
```

A.5 CLEAR

Bits of a byte may be set to 0 according to a specified mask.

```
                BITS/     [variable]
<label:> CLEAR                    OF name;
                BIT
                or
<label:> CLEAR condition code;
```

The mask may be a variable name (unsubscripted) or an integer. The target must be an unsubscripted variable name. All of the bit positions which are 1 in the binary representation of the mask are set to zero in the target. Condition codes may also be cleared with this instruction.

Examples:

```
CLEAR BITS[5] OF B;
CLEAR BIT[4] OF B;
CLEAR BIT[200] of D;
CLEAR BITS[ALPHA] OF BETA;
" CONDITION CODES ";
CLEAR .CARRY;
CLEAR .DECIMAL;
CLEAR .INTERRUPT;
```

A.6. CODE

Assembler language code may be inserted directly into programs.

```
<label:>  <CODE>  'ASSEMBLER STATEMENT';
```

The statement between quotes is passed directly to the output data set exactly as it is specified. The keyword CODE is optional. The code statement is useful for specifying assembly language directives not in the PL/65 language. The code statement should be used with due caution and the user should be familiar with the code produced by the compiler. Blocks of code may be passed between two lines which have an '*' in the first column.

Examples:

```
CODE ' LDA #5';
' BCC *+5';
*START OF BLOCK OF CODE
LDA #5
STA BB
LDX #$6A
TXS
*END OF ASSEMBLER CODE
```

A.7 COMMENT

Comments may be used freely wherever statements can occur.

```
"characters" ;
```

Comments are delimited by double quotes and terminated by a semicolon.

Examples:

```
"THIS IS A SAMPLE STAND ALONE COMMENT";
B = C + 1; "INCREMENT C, ASSIGN TO B";
```

A.8 DATA

Arrays of integers or character strings can be initialized with the DATA statement.

```
                              expr/
<label:>          DATA<W>     list/    ;
                              'characters'
```

An integer list is a series of integer constants separated by commas. A list of variable names can also be used; however, the value represented by the symbol cannot exceed 255 in the DATA statement. A label is optional, but required for all practical purposes for referencing the array. Since the data statement is an implied declaration the label symbol cannot also appear in a declaration. Note that a character list may not be used with the DATAW command.

Examples:

```
B: DATA 2, 3, 6;
DOG: DATA C, BETA;
IOTA: DATA 'READ'; "CHARACTER STRING CONSTANT";
DATAW 2,3,6;
DATAW C,BETA;
DATA BETA, BETA+40, BETA+$40;
DATAW BETA,BETA+20,BETA+40;
```

A.9 DEC<W>

Decrement the value represented by a symbol.

```
<label:>        DEC<W>    name;
```

This statement corresponds to the decrement statement common to microprocessors. An alternate form for decrementing bytes is also shown in the examples.

Examples:

```
DEC ALPHA;
ALPHA - 1; "SAME AS ABOVE";
DECW ALPHA; "WORD DECREMENT";
```

NOTE:

It is sometimes necessary to decrement a word that is a decimal number. One way to do this is to declare a word that has a value of one and subtract that from the variable with the decimal flag on. Example:

```
DCL ONE WORD INIT[1];
DCL NUMB WORD;
...
```

```
SET .DECIMAL;
NUMB.2 = NUMB-ONE;
   or
NUMB == NUMB-ONE;
CLEAR .DECIMAL;
```

A.10 DECLARE

Declarations are used to reserve storage for bytes, pairs of bytes (words), and linear arrays of bytes of words.

```
DCL              BYTE/   INIT<IAL> [variable];
       name      WORD/   [variable];
DECLARE          CHAR<ACTER>['characters'];
```

Words, bytes, and character strings can be data initialized. Allowable keyword abbreviatins are DCL for DECLARE, CHAR for CHARACTER, and INIT for INITIAL.

Examples:

```
DCL ALPHA; "RESERVE ONE BYTE":
DCL B WORD; "RESERVE ONE WORD [TWO BYTES]";
DCL D CHAR; "RESERVE A BYTE";
DCL E BYTE INIT[5]; "RESERVE A BYTE WITH VALUE 5";
DCL F WORD INIT[44]; "RESERVE A WORD WITH VALUE 44";
DCL H CHAR['#']; "RESERVE A BYTE FOR CHARACTER #";
DCL J[5]; "RESERVE A LINEAR ARRAY OF 5 BYTES";
DCL K CHAR['LOST DATA']; "INITIALIZE A STRING";
DCL BUF[80]; "DEFINE AN ARRAY OF 80 BYTES";
```

A.11 DEFINE

A name may be defined to have a value (i.e., address) as in assembly language equates.

```
                DEF /
<label:>                    name=expr,name=exr,..;
                DEFINE
```

There is almost no syntax checking done by the compiler on the DEFINE expression. Any string of symbols is accepted by the compiler without error. Errors in the expression (if any) are flagged at assembly time rather than at translate time.

Examples:

```
 DEF M = 25; "THE SYMBOL M HAS THE VALUE 25";
"NOTE THIS IS NOT THE SAME AS ASSIGNMENT OF";
"THE VALUE 25 TO THE NAME M";
 DEF N = *; "THE SYMBOL N IS GIVEN THE VALUE OF PC";
 DEF P = N + 5; "N IS REQUIRED TO BE ALREADY DEFINED";
 DEF * = * + 4; "ADVANCE THE INSTRUCTION COUNTER";
 DEFINE AA=3, BB=AA+3, JJ=AA+BB-2;
```

A.12 ENTRY

The assembler instruction counter can be explicitly set.

```
ENTRY <variable> ;
```

The integer specifies the value to be taken by the assembler instruction counter. This command also generates initialization code for clearing decimal mode and establishing the stack pointer at $FF. Normally the first statement of a PL/65 program will be the ENTRY statement, though it can be used at any point.

Examples:

```
ENTRY;
ENTRY 1000;
ENTRY 256; "VALUES ARE DECIMAL";
ENTRY $6A00;
ENTRY START;
```

A.13 EXIT

Termination of a PL/65 program is made by EXIT.

```
<label:>     EXIT;
```

This statement generates the .END assembler directive. Normally the last statement of a PL/65 program will be the EXIT statement.

Examples:

```
EXIT;
L1: EXIT;
```

A.14 FOR-TO-BY

Looping with iteration is implemented with the FOR-loop.

```
<label:> FOR name = expl TO variable2 <BY variable 3>

                        block ;
```

First expression 1 (expl) is computed and the value assigned to the name specified. Next the value of expl is compared with variable2. If the value of expl is less than or equal to variable2 the block is executed. Then the value of variable3 is added to the value for "name" and the process is repeated with expl being recomputed. If the BY clause is absent the value 1 is assumed for variable3. Note that the block might not be executed at all depending on expl and variable2.

Examples:

```
 FOR J = 1 TO N
BEGIN;
 ...
END;
 FOR ALPHA = BETA+3 TO GAMMA BY DELTA
BEGIN;
 ...
END;
FOR B = H[I+J .AND K] + L .OR M TO M
 BEGIN;
 ...
 END;
FOR B=1 TO H BY -2
 BEGIN;
 ...
 END;
 FOR B = 1 TO 25  H[B]=0;
```

A.15 GO TO

Absolute branching can be made to any labeled statement.

```
<label:>  GO TO  <@>name ;
```

Normal sequential execution can be interrupted and a branch made to the label indicated. There are no restrictions on branching out of blocks or iteration groups. Computed GOTO is covered under the CASE statement.

Examples:

```
GO TO ALPHA;
GO TO @ALPHA;
GO TO L5;
BEGIN;
IF  B = C THEN GO TO LAB2;
C = D + 2;
END;
```

A.16 HALT

Execution is permanently terminated.

```
<label:> HALT ;
```

PL/65 generates a jump-to-self instruction (i.e., JMP *).

Examples:

```
 HALT;
L5: HALT;
```

A.17 IF-THEN-ELSE

This statement form makes possible the conditional execution of sections of code based on testing a relational expression.

```
                 expl relopr exp2   statement;/  statement;/
<label:> IF      BIT<S> OF name     THEN         <ELSE>
                 condition code       block
```

If the relation is true the THEN clause is executed, otherwise the ELSE clause is executed. The ELSE clause is always optional. The relopr (relational operator) must be chosen from:

SINGLE BYTE

< less than	<= less than or equal to
> greater than	>= greater than or equal to
= equal to	^= not equal to

MULTIPLE BYTE

= equal to	^= not equal to

Examples:

```
 IF B < C THEN
D = E;
ELSE
F = G;
 IF B-5 = C-D THEN
 BEGIN;
B = B + 1;
C = D;
END;
 IF E > F THEN
CALL SUB1;
ELSE
BEGIN;
WAIT ON BIT[1] OF TS;
B[B4] = C[3];
END;
"INDIRECT COMPARISONS ARE ALSO POSSIBLE";
 IF @B=3 THEN RTI;
```

```
 IF B=3 THEN RTI;
 IF @B[J]=5 THEN RETURN;
 IF B[5]=J THEN RTI;
 IF @B[5]>J THEN RETURN;
 IF B[6] < J THEN GOTO B;
"CAN TEST BITS":
 IF BIT[2] OF BB THEN RETURN;
 IF BITS[BB] OF C THEN GOTO B;
"MULTIPLE BYTE COMPARISONS";
 IF B[K].J = D THEN RTI;
 IF B[K].J^ = D THEN
        DO;
        ...
        END;
 ELSE
        DO;
        ...
        END;
"CONDITION CODES";
 IF .ZERO THEN GOTO B;
 IF^ = THEN B=2;
```

A.18 IFF-THEN

This is an alternate form of conditional execution which corresponds to compare and conditional branch in the R6500.

```
                    named register relopr expl
<label:> IFF        BIT[variable]   THEN <GOTO> label;
                    operator
                    condition code
```

The named register must be .A for accumulator, .X for index X, or .Y for index Y. If the relation is true then a branch is made to the label indicated. The relopr (relational operator) is more restrictive than that in the IF-THEN-ELSE statement and must be:

```
<   less than
>=  greater than or equal to
=   equal to
```

The condition code must be chosen from among:

```
.C Carry
```

```
.O Overflow
.N Negative
.Z Zero
```

The complement of each of these condition codes can be tested by using a preceding the name. The operator can be any of +, -, or =. Note that the 'GO TO' after the 'THEN' is optional. The named register must be chosen from:

```
.A = Accumulator
.X = Index X
.Y = Index Y
```

Examples:

```
IFF .A = 5 THEN L4;
IFF .C THEN START;
IFF ^.O THEN ALPHA;
IFF + THEN DONE;
IFF - THEN BACK;
IFF = THEN BETA;
IFF ^= THEN L2;
IFF BIT[4] THEN L2;
IFF BIT[ALPHA] THEN GO TO L3;
IFF .NOT= THEN BETA;
IFF .NOT .CARRY THEN GO TO BETA;
IFF .NOT .OVERFLOW THEN BETA;
IFF .NOT .ZERO THEN BETA;
IFF = THEN BETA;
IFF .CARRY THEN BETA;
IFF .OVERFLOW THEN BETA;
IFF .ZERO THEN BETA;
IFF .NEGATIVE THEN BETA;
IFF .X=5 THEN JAMIT;
IFF .Y<6 THEN JAMIT;
IFF .A>=7 THEN BETA;
IFF .X^=3 THEN BETA;
```

A.19 INC<W>

Increment the value represented by a symbol.

```
<label:> INC<W> name ;
```

This statement corresponds to the increment statement common to microprocessors. An alternate form for incrementing bytes is also shown in the examples.

Examples:

```
INC BETA;
BETA + 1; "SAME AS ABOVE";
INCW BB;
```

It is sometimes useful to be able to increment a decimal word. The following sequence is one way of doing this.

```
DCL ONE WORD INIT[1];
DCL NUMB WORD;
 ...
SET .DECIMAL;
NUMB == NUMB+ONE;
CLEAR .D;
```

A.20 PULSE

Bits may be turned ON-OFF-ON or OFF-ON-OFF.

```
                        BIT /           SET/
<label:>     PULSE              [variable]      OF name;
                        BITS            CLEAR/CLR
```

The mask may be a variable name (unsubscripted) or an integer. The target must be an unsubscripted variable name. The SET condition is default. It assumes the bits are ones and the instruction changes the bits to zeroes then back to ones. Alternatively the CLR specification assumes the bits are zeroes, changes them to ones and back to zeroes again.

Examples:

```
PULSE BIT[1] OF BETA;
PULSE BIT[1] SET OF BETA; "SAME AS ABOVE";
PULSE BITS[ALPHA] CLR OF BETA;
```

A.21 RETURN

Subroutine exits are made via a RETURN statement.

```
<label:>  RETURN ;
```

A subroutine call saves the return address to be used when a return is executed. The RETURN signifies that execution is to be resumed at the statement following the most recent call.

Examples:

```
CALL SUB1;
...
SUB1: BEGIN;
...
RETURN;
END;
```

A.22 ROTATE

Bytes may be rotated left or right (end-around fill).

```
                LEFT/
<label:> ROTATE         name<[sub]><.expr>
                RIGHT
```

The specified byte is rotated. The integer specifies the number of bit positions for the rotation. The abbreviations ROL and ROR can be used for ROTATE LEFT and ROTATE RIGHT.

Examples:

```
ROTATE LEFT Q.4;
ROL Q.4;
ROR R;
ROTATE RIGHT R.1;
"CAN ALSO USE SUBSCRIPTS";
ROR B[Q];
```

```
ROR B[Q-5];
ROL B[Q-5].3;
"QUANTIFIER MAY BE EXPRESSION";
ROL B[K+5-J].Q-J+3;
```

A.23 RTI

A return from interrupt may be explicitly specified by the user.

```
<label:>  RTI ;
```

This statement causes a return from an external or BRK interrupt. It may be used in an interrupt service routine block to specify alternate exit points from the block.

Examples:

```
BEGIN;
...
RTI;
...
L1: RTI;
...
END;
```

A.24 SET

Bits of a byte may be set to 1 according to a specified mask.

```
                BIT/
<label:> SET            [variable]     OF name;
                BITS
                or
<label:> SET    condition code;
```

The mask may be a variable name (unsubscripted) or an integer. The target must be an unsubscripted variable name. All of the bit positions which are 1 in the binary representation of the mask are set to 1 in the target. Condition codes may also be explicitly set with this command.

Examples:

```
SET BITS[5] OF D;
SET BIT[4] OF B;
SET BIT[158] OF C;
SET BIT[255] OF D;
" ALSO CAN SET CONDITION CODES ";
SET .DECIMAL;
SET .D;
SET .CARRY;
SET .C;
SET .INTERRUPT;
SET .I;
SET .DECIMAL;
SET .D;
```

A.25 SHIFT

Bytes may be shifted left or right (zero fill).

```
                  LEFT/
<label:> SHIFT          name<[sub]><.expr>;
                  RIGHT
```

The specified byte is shifted. The integer specifies the number of bit positions for the shift. The abbreviations SHL and SHR can be used for SHIFT LEFT and SHIFT RIGHT.

Examples:

```
SHIFT LEFT B.3;
SHL B.3;
SHR C;
SHIFT RIGHT C.3;
"CAN ALSO SUBSCRIPT AND QUANTIFY WITH EXPR";
SHR B[K+5 .AND 1].M-K+3;
```

A.26 STACK

Variables may be stored on a stack on a last-in first-out basis.

```
<label:> STACK <WORD> namelist;
```

The hardware byte stack of the R6500 is used by this instruction and hence is limited to 256 bytes of memory. It is convenient for storing register values on entry to subroutines. Multiple names separated by commas are permitted. Note that word variables may also be stacked.

Examples:

```
STACK .A, .X, .Y;
STACK BETA, GAMMA;
STACK WORD PC;
STACK 5;
STACK -5;
STACK $AB;
```

A.27 TAB

The TAB character is used to format the LISTING file of PL/65 source code for better readability. A 'TAB' character may always be used. On the SYSTEM 65, a '/' may also be used, since TAB is not supported in the SYSTEM 65 Editor.

A.28 UNSTACK

Bytes may be fetched from a stack on a last in-first out basis.

```
<label:> UNSTACK <WORD> namelist;
```

Items are retrieved from a stack in reverse order of how they were stacked. Multiple names are separated by commas. Either bytes or words may be unstacked.

Examples:

```
UNSTACK .Y, .X, .A;
UNSTACK GAMMA, BETA;
UNSTACK WORD PC;
```

A.29 WAIT

Execution is temporarily suspended awaiting external events.

```
                ANY/ BITS/        SET/
<label:>WAIT ON            [variable] OF name;
                ALL BIT           CLEAR/CLR
```

Execution resumes with the instruction following the WAIT as soon as the WAIT condition is satisfied. The default options are ALL bits SET (i.e., set to 1).

Examples:

```
WAIT ON BITS[ALPHA] OF BETA;
WAIT ON ANY BITS[ALPHA] SET OF BETA; "SAME AS ABOVE";
WAIT ON ANY BITS[ALPHA] CLEAR OF BETA;
WAIT ON ALL BITS[ALPHA] SET OF BETA;
WAIT ON ALL BITS[ALPHA] CLEAR OF BETA;
WAIT ON BITS[5] OF BETA;
WAIT ON BITS[5] CLEAR OF BETA;
```

A.30 WHILE

The WHILE-loop makes possible the conditional execution of a block based on the computed value of an expression.

```
                                  statement
<label:> WHILE expl relopr variable
                                  /BLOCK
```

If the relation is true then the block is executed. Following execution of the block the sequence is repeated with evaluation of the relation. Repeated continuous execution of the loop is possible by specifying a relation which is always true.

Examples:

```
WHILE X + 1 < Y
DO;
...
END;
WHILE 2=2
; "THIS IS A NON-TERMINATING LOOP";
DO;
...
END;
```

APPENDIX B

ABBREVIATIONS

The following is a list of the permissable relationships, operators, registers and their abbreviations (if any)

1.	AND	=	& or .AND
2.	BIT	=	BIT or BITS
3.	CLEAR	=	CLR
4.	DECIMAL	=	.DECIMAL or .D
5.	INITIAL	=	INIT or INITIAL
6.	INTERRUPT	=	INTERRUPT or .I
7.	OVERFLOW	=	.OVERFLOW or .O
8	OR	=	.OR or
9.	NEGATIVE	=	.NEGATIVE or .N
10.	EXCLUSIVE OR	=	.EOR or .XOR
11.	ZERO	=	.ZERO or .Z or =
12.	ACCUMULATOR	=	.A
13.	INDEX X	=	.X
14.	INDEX Y	=	.Y
15.	STACK POINTER	=	.S
16.	PROCESSOR STATUS	=	.P
17.	CARRY	=	.CARRY or .C
18.	HEXADECIMAL	=	$
19.	BINARY	=	%

APPENDIX C

ASSEMBLER EQUIVALENTS

There are a number of PL/65 constructs which are exactly equivalent to a single assembler language statement. The following table lists the assembler statement and its PL/65 equivalent.

ASSEMBLER	**PL/65**
LDA #4	.A=4;
LDA #FIX	.A=#FIX;
LDA BETA	.A=BETA;
LDA BETA,X	.A=BETA[.X];
LDA BETA,Y	.A=BETA[.Y];
LDA (BETA,X)	.A=@BETA[.X];
LDA (BETA),Y	.A=@BETA[.Y];
CLC	CLEAR .C;
CLD	CLEAR .D;
CLI	CLEAR .I;
CLV	CLEAR .O;
SEC	SET .C;
SED	SET .D;
SEI	SET .I;
ASL BETA	SHIFT LEFT BETA; (or SHL BETA;)
LSR BETA	SHIFT RIGHT BETA; (or SHR BETA;)
ROL BETA	ROTATE LEFT BETA; (or ROL BETA;)
ROR BETA	ROTATE RIGHT BETA; (or ROR BETA;)
INC BETA	BETA+1; (or INC BETA;)
DEC BETA	BETA-1; (or DEC BETA;)
INX	.X+1; (or INC .X;)
DEX	.X-1; (or DEC .X;)
INY	.Y+1; (or INC .Y;)
DEY	.Y-1; (or DEC .Y;)
BCC L1	IFF .CARRY THEN L1;
BCS L1	IFF .CARRY THEN L1;
BVC L1	IFF .OVERFLOW THEN L1;
BVS L1	IFF .OVERFLOW THEN L1;
BEQ L1	IFF = THEN L1;
BNE L1	IFF = THEN L1;

ASSEMBLER	**PL/65**
BPL L1	IFF + THEN L1;
BMI L1	IFF - THEN L1;
JMP BETA	GO TO BETA;
JMP (BETA)	GO TO @BETA;
JSR BETA	CALL BETA;
RTS	RETURN;
RTI	RTI;
BRK	BRK; (or BREAK;)
PLA	UNSTACK .A;
PHA	STACK .A;

APPENDIX D

SAMPLE PL/65 PROGRAMS

D.1 "SORT" COMPILER INPUT

```
0001; PAGE '**SORT**';
0002; ;
0003; 'ASCENDING ORDER SORT';
0004; ;
0005; DECLARE F, I,TMP;
0006; ENTRY $200;
0007; ;
0008; N=N-2; 'SET TERMINAL VALUE FOR LOOP';
0009; F = 1; 'SET FLAG';
0010; WHILE F = 1
0011;       DO;
0012;       F = 0;
0013;           FOR I = 0 TO N
0014;           BEGIN;
0015;           IF B[I] > B[I+1] THEN
0016;               BEGIN;
0017;               F = 1;
0018;               TMP = B[I];
0019;               B[I] = B[I+1];
0020;               B[I+1] = TMP;
0021;               END;
0022;           END;
0023;       END;
0024; N: DATA 10; 'N IS NUMBER OF SORT ELEMENTS';
0025; B: DATA 23,55,36,28,54,39,99,86,21,67;
0026; ;
0027; EXIT;
```

D.2 "SORT" COMPILER OUTPUT

```
; PAGE '**SORT**';
.PAG '**SORT**'
; ;
; "ASCENDING ORDER SORT";
; ;
; DECLARE F, I,TMP;
F   *=*+1
I   *=*+1
TMP   *=*+1
; ENTRY $200;
R0=0
R1=2
R2=3
R3=4
 *=$200
CLD
LDX #$FF
TXS
; ;
; N=N-2; "SET TERMINAL VALUE FOR LOOP";
LDA N
SEC
SBC #2
STA N
; F = 1; "SET FLAG";
LDA #1
STA F
; WHILE F = 1
ZZ0001 LDA F
CMP #1
BEQ  *+5
JMP ZZ0002
;   DO;
;   F = 0;
LDA #0
STA F
;    FOR I = 0 TO N
LDA #0
STA I
ZZ0003 CMP N
BEQ *+7
BCC *+5
JMP ZZ0004
;    BEGIN;
;    IF B(I) > B(I+1) THEN
```

```
LDA I
TAY
LDA I
CLC
ADC #1
TAX
LDA B,X
CMP B,Y
BCC  *+5
JMP ZZ0005
;     BEGIN;
;     F = 1;
LDA #1
STA F
;     TMP = B[I];
LDA I
TAX
LDA B,X
STA TMP
;     B[I] = B[I+1];
LDA I
TAY
LDA I
CLC
ADC #1
TAX
LDA B,X
STA B,Y
;     B[I+1] = TMP;
LDA I
CLC
ADC #1
TAY
LDA TMP
STA B,Y
;     END;
;    END;
ZZ0005
INC I
LDA I
JMP ZZ0003
ZZ0004
;   END;
JMP ZZ0001
ZZ0002
; N: DATA 10; "N IS NUMBER OF SORT ELEMENTS";
N
.BYT 10
; B: DATA 23,55,36,28,54,39,99,86,21,67;
B
.BYT 23,55,36,28,54,39,99,86,21,67
; ;
; EXIT;
.END
```

D.3 "TOGGLE TEST" COMPILER INPUT

```
0001; PAGE 'TOGGLE TEST';
0002; ;
0003; DEF DRB=$A000;
0004; DEF T1CL =$A004, T1CH =$A005;
0005; DEF ACR =$A00B, PCR=$A00C;
0006; DEF IFR=$A00D;
0007; DEFINE *=$10;
0008; DCL J,K,L;
0009; DCL TIME;
0010; ENTRY;
0011;       DDRB=$FF;
0012; ST: CLEAR .CARRY;
0013;       T1CL=$50;
0014;       T1CH=$C3;
0015;       FOR J=0 TO 10 BY 2
0016;           BEGIN;
0017;           DRB=T[J];
0018;           TIME=T[J+1];
0019;           CALL DELAY;
0020;           END;
0021; GO TO ST;
0022; ;
0023; DELAY: FOR K=1 TO TIME
0024;       BEGIN;
0025;           'FOLLOWING IS A 1 SECOND DELAY';
0026;           FOR L=1 TO 20
0027;           BEGIN;
0028;           WHILE IFR ^=0;
0029;           END;
0030;       END;
0031; RETURN;
0032; ;
0033; 'FOLLOWING IS SWITCH POSITION AND TIME DELAY';
0034; DATA 1,5,0,10,1,5,0,15,1,5,0,2;
0035; EXIT;
```

D.4 "TOGGLE TEST" COMPILER OUTPUT

```
; PAGE 'TOGGLE TEST';
.PAG 'TOGGLE TEST'
; ;
; DEF DRB=$A000;
DRB=$A000
; DEF T1CL =$A004, T1CH =$A005;
T1CL=$A004
T1CH=$A005
; DEF ACR =$A00B, PCR=$A00C;
ACR=$A00B
PCR=$A00C
; DEF IFR=$A00D;
IFR=$A00D
; DEFINE *=$10;
*=$10
; DCL J,K,L;
J   *=*+1
K   *=*+1
L   *=*+1
; DCL TIME;
TIME   *=*+1
; ENTRY;
R0=0
R1=2
R2=3
R3=4
 *=$200
CLD
LDX #$FF
TXS
;   DDRB=$FF;
LDA #$FF
STA DDRB
; ST: CLEAR .CARRY;
ST
CLC
;   T1CL=$50;
LDA #$50
STA T1CL
;   T1CH=$C3;
LDA #$C3
STA T1CH
;   FOR J=0 TO 10 BY 2
LDA #0
```

```
STA J
ZZ0001 CMP #10
BEQ *+7
BCC *+5
JMP ZZ0002
;    BEGIN;
;    DRB=T[J];
LDA J
TAX
LDA T,X
STA DRB
;    TIME=T[J+1];
LDA J
CLC
ADC #1
TAX
LDA T,X
STA TIME
;    CALL DELAY;
JSR DELAY
;    END;
LDA #2
CLC
ADC J
STA J
JMP ZZ0001
ZZ0002
; GO TO ST;
JMP ST
; ;
; DELAY: FOR K=1 TO TIME
DELAY
LDA #1
STA K
ZZ0003 CMP TIME
BEQ *+7
BCC *+5
JMP ZZ0004
;   BEGIN;
;    'FOLLOWING IS A 1 SECOND DELAY';
;    FOR L=1 TO 20
LDA #1
STA L
ZZ0005 CMP #20
BEQ *+7
BCC *+5
JMP ZZ0006
;    BEGIN;
;    WHILE IFR ^=0;
ZZ0007 LDA IFR
```

```
CMP #0
BNE  *+5
JMP ZZ0008
JMP ZZ0007
ZZ0008
;    END;
INC L
LDA L
JMP ZZ0005
ZZ0006
;   END;
INC K
LDA K
JMP ZZ0003
ZZ0004
; RETURN;
RTS
; ;
; "FOLLOWING IS SWITCH POSITION AND TIME DELAY";
; DATA 1,5,0,10,1,5,0,15,1,5,0,2;
.BYT 1,5,0,10,1,5,0,15,1,5,0,2
; EXIT;
.END
```

D.5 "MONITOR SEGMENT" COMPILER INPUT

```
0001; PAGE 'MONITOR SEGMENT';
0002; ;
0003; "SEGMENT OF A MONITOR PROGRAM";
0004; RST:  .X=$FF;
0005;       .S=.X
0006;       SPUSER=.X;
0007;       CALL INITS;
0008; DETCPS: CNTH30=$FF;
0009;       .A=$01;
0010; DET1: IFF BIT[SAD] THEN START;
0011;       IFF .N THEN DET1;
0012;       .A=$FC;
0013; DET3: CLEAR .CARRY;
0014;       .A+$01;
0015;       IFF ^.C THEN DET2;
0016;       CNTH30+1;
0017; DET2: .Y=SAD;
0018;       IFF ^.N THEN DET3;
0019;       CNTH30=.A;
0020;       .X=$08;
0021;       CALL GET5;
0022; EXIT;
```

D.6 "MONITOR SEGMENT" COMPILER OUTPUT

```
; PAGE 'MONITOR SEGMENT';
.PAG 'MONITOR SEGMENT'
; ;
; "SEGMENT OF A MONITOR PROGRAM";
; RST: .X=$FF;
RST
LDX #$FF
;   .S=.X
TXS
;   SPUSER=.X;
STX SPUSER
;   CALL INITS;
JSR INITS
; DETCPS: CNTH30=$FF;
DETCPS
LDA #$FF
STA CNTH30
;   .A=$01;
LDA #$01
; DET1: IFF BIT[SAD] THEN START;
DET1
BIT SAD
BNE START
;   IFF .N THEN DET1;
BMI DET1
;   .A=$FC;
LDA #$FC
; DET3: CLEAR .CARRY;
DET3
CLC
;   .A+$01;
CLC
ADC #$01
;   IFF ^.C THEN DET2;
BCC DET2
;   CNTH30+1;
INC CNTH30
; DET2: .Y=SAD;
DET2
LDY SAD
;   IFF ^.N THEN DET3;
BPL DET3
;   CNTH30=.A;
STA CNTH30
;   .X=$08;
LDX #$08
;   CALL GET5;
JSR GET5
; EXIT;
.END
```

APPENDIX E

PL/65 TEST PROGRAM

```
0001; PAGE '**TEST  CASES**';
0002; ' COMPILED ON SYSTEM 65 ';
0003; DCL B,L,G,C,GAMMA,DELTA,Q,R,BETA;
0004; ENTRY $220;
0005; ;
0006; ' PL/65 TEST CASES FROM MANUAL ';
0007; ;
0008; '*** ASSIGN ***';
0009; ;
0010; B=C;
0011; @B = C;
0012; B = @C;
0013; B = $C;
0014; C = D + F;
0015; C = D-F;
0016; C = D .AND F;
0017; C = D .OR F;
0018; C = D .XOR F;
0019; C = D .AND F .OR 2 .XOR $55;
0020; BB = C .OR D;
0021; BB = C .XOR D;
0022; BB = C .AND D;
0023; BB .AND 1;
0024; BB .OR 1;
0025; BB .XOR 1;
0026; ' ABBREVIATED FORM ';
0027; B+1;
0028; B-1;
0029; B+J;
0030; B-J;
0031; B+J-K+5;
0032; F=G[4];
0033; B[J] = C;
0034; B[J+5] = C;
0035; ' SINGLE BYTE INDIRECTS'
0036; @B[K] = C[5]+D-F;
0037; &B[K] = C[5]+D-F;
0038; &B = C[R]+D-6 .AND 3;
0039; &B[G-3] = C[N]+D-6;
0040; &B[3] = C[4]-(F-D+3);
0041; B[5] = @C+D-(F-N+3);
0042; B[5] = @C;
0043; B[N] = &C;
0044; @B=@C;
0045; &B=&C;
0046; &B=@C;
0047; @B=&C;
0048; @B[1]=@C[2];
0049; &B[1]=&C[2];
0050; &B[1]=@C[2];
0051; @B[1]=&C[2];
0052; ' MULTI BYTE ARITHMETIC ';
0053; B.K = C;
0054; B[J+1].K-1 = 'ABCDEF' ;
0055; B[J+1].K = C[K];   ' NO ADDITIONAL TERMS WITH';
0056; '    RIGHT SUBSCRIPT ';
0057; B.K = C[J];
0058; B.K = C-D; 'ONLY TWO TERMS IN MULTI BYTE ';
```

```
0059; B[J].15 = C[J];
0060; B.K = 5; "SET K BYTES ALL EQUAL TO 5 ";
0061; B[J-K+1].K-1 = DELTA -GAMMA;
0062; " PARENTHETICAL GROUPING ";
0063; B = G -(C+GAMMA-2) ;
0064; B = G+(C-GAMMA+2);
0065; B = G+DELTA-(GAMMA-C+3)-$55;
0066; B = G+(GAMMA - (DELTA-2)+6) ;
0067; B = G-(DELTA+(GAMMA-3));
0068; " NOTE NO ( MAY BE USED IMMEDIATELY AFTER = SIGN ";
0069; ;
0070; "*** BLOCK ***";
0071; ;
0072; FOR J = 1 TO 5
0073;     DO;
0074;     B[J] = J;
0075;     C[J] = J-1;
0076;     END;
0077; ;
0078; IF ALPHA = 0 THEN
0079;     BEGIN;
0080;     B = $21;
0081;     C = 17;
0082;     D = %1011;
0083;     END;
0084; ;
0085; "*** BREAK ***";
0086; BRK;
0087; ;
0088; "*** CALL ***";
0089; ;
0090; CALL SUB1;
0091; CALL ALPHA;
0092; SUB1: BEGIN;
0093; ;
0094;         RETURN;
0095;         END;
0096; ;
0097; ALPHA: BEGIN;
0098;          ;
0099;          RETURN;
0100;          END;
0101; ;
0102; "*** CASE ***";
0103; CASE [N][L1,L2,L3];
0104; GO TO [N][L1,L2,L3];
0105; GO TO [B] [ALPHA,BETA];
0106; GOTO [N+B-1] [ALPHA,BETA,L1,L2,L3];
0107; CASE [ B[J+C .OR 40] +N .AND $3F] [L1,L2,L3];
0108; ;
0109; "*** CLEAR ***";
0110; ;
0111; CLEAR BITS[5] OF B;
0112; CLEAR BIT[4] OF B;
0113; CLEAR BIT[200] OF D;
0114; CLEAR BITS[ALPHA] OF BETA;
0115; ;
0116; " ALSO CAN CLEAR CONDITIONAL CODES ";
```

```
0117; " .CARRY OR .C"; " .DECIMAL OR .D ";
0118; " .INTERRUPT OR .I "; " .OVERFLOW OR .O ";
0119; CLEAR .C; CLEAR .D;
0120; CLEAR .INTERRUPT; CLEAR .OVERFLOW; CLEAR .O;
0121; ;
0122; "*** CODE ***";
0123; ;
0124; CODE ' .PAGE ';
0125; ' BCC *+5';
0126; " BLOCKS OF ASSEMBLER CODE MAY ALSO BE PASSED..";
0127; "...BETWEEN TWO LINES WITH A '*' IN FIRST COL ";
0128; * START OF ASSEMBLER CODE
0129; LDA #5
0130; STA BB
0131; LDX #$6A
0132; TXS
0133; ; ASSEMBLER COMMENT
0134; INY
0135; STA B,Y  ;COMMENT
0136; * END OF ASSEMBLER CODE
0137; ;
0138; "*** COMMENTS ***";
0139; ;
0140; B = C+1 ; "INCREMENT C, ASSIGN TO B";
0141; ;
0142; "*** DATA ***";
0143; ;
0144; BBC: DATA 2,3,6;
0145; DOG: DATA C,BETA;
0146; IOTA: DATA 'READ';
0147; DATA BETA, BETA+40, BETA+$40;
0148; "*** DATA WORD***";
0149; WDOG: DATAW 1,2,BBC;
0150; DATAW #BETA, BETA+20, BETA+60;
0151; ;
0152; "*** DEC ***";
0153; ;
0154; DEC ALPHA;
0155; ALPHA-1;
0156; ;
0157; "*** DECREMENT WORD***";
0158; DECW ALPHA;
0159; ;
0160; "*** DECLARE ***";
0161; ;
0162; DCL ALP,I;
0163; DCL BB WORD;
0164; DCL D CHAR;
0165; DCL E BYTE INIT[5];
0166; DCL FTH WORD INIT[44];
0167; DCL H CHAR['#'];
0168; DCL J[5];
0169; DCL K CHAR['LOST DATA'];
0170; DCL BUF[80];
0171; DCL ONE WORD INIT[1];
0172; ;
0173; "*** DEFINE ***";
0174; ;
```

```
0175; DEF M = 25;
0176; DEF N=*;
0177; DEF PP= N+5;
0178; DEF * = * +4;
0179; DEF AA=3, BBF=AA+3, JJ=AA+BBF-2;
0180; DEF GT = -BB+$33;
0181; ;
0182; '*** ENTRY ***';
0183; ;
0184; ENTRY;
0185; ENTRY $1000;
0186; ;
0187; '*** FOR-TO-BY ***';
0188; ;
0189; FOR J = 1 TO N
0190;   BEGIN;
0191;    END;
0192; FOR ALPHA = BETA+3 TO GAMMA BY DELTA
0193;    BEGIN;
0194;    END;
0195; FOR B = H[I+J .AND K] +L .OR M TO 6
0196; BEGIN; END;
0197; FOR B=1 TO H BY -2
0198; BEGIN; END;
0199; ;
0200; '*** GO TO ***';
0201; ;
0202; GO TO ALPHA;
0203; GOTO @ALPHA; 'INDIRECT JUMP';
0204; GOTO L5;
0205; BEGIN;
0206;   IF B = C THEN GO TO LAB2;
0207;    C = D + 2;
0208; END;
0209; ;
0210; '*** HALT ***';
0211; ;
0212; HALT;
0213; LAB99:   HALT;
0214; ;
0215; '*** IF-THEN-ELSE ***';
0216; ;
0217; IF B < C THEN
0218;    D = E ;
0219; ELSE
0220;    F = G ;
0221; IF B-5 = C-D THEN
0222;    BEGIN;
0223;    B = B + 1;
0224;    C = D;
0225;    END;
0226; ELSE
0227;    BEGIN;
0228;    WAIT ON BIT[4] OF TS;
0229;    B[4] = C[3];
0230;    END;
0231; ' CONDITION CODES MAY BE TESTED ALSO ';
0232; ' VALID TESTS ARE:  ';
```

```
0233;  " .ZERO OR .Z "; " .CARRY OR .C ";
0234;  " .OVERFLOW OR .O "; " .NEGATIVE OR .N ";
0235;  " = ";
0236;  " OR THE .NOT OR ^ OF ANY ABOVE ";
0237;  IF .ZERO THEN GOTO B;
0238;  IF .NOT .Z THEN GOTO B;
0239;  IF ^ .ZERO THEN RETURN;
0240;  IF = THEN B=1;
0241;  IF ^= THEN B=2;
0242;  IF B=AA+BB-3 THEN BETA+1;
0243;  IF 'AB'.2 = B THEN GOTO OUT;
0244;  IF 'A' = B THEN C=2;
0245;  IF 'A'.1 ^= B THEN C=3;
0246;  IF 'G' = B+C THEN RETURN;
0247;  IF B[3] = G+H THEN B=5;
0248;  IF B[K].J = D THEN RETURN;
0249;  " INDIRECT COMPARISONS";
0250;  IF @R=3 THEN RTI;
0251;  IF &B=3 THEN RTI;
0252;  IF @B[J]=5 THEN RETURN;
0253;  IF &B[5]=J THEN RTI;
0254;  IF @B[5]>J THEN RETURN;
0255;  IF &B[J]<6 THEN RETURN;
0256;  " CAN TEST BITS ";
0257;  IF BIT[2] OF BB THEN RETURN;
0258;  IF BITS[BB] OF C THEN GOTO B;
0259;  IF BIT[2] OF BB THEN RETURN;
0260;  IF BITS[5] OF BB THEN AA=3;
0261;  IF BITS[BB] OF C THEN GO TO B;
0262;  IF BITS[$80] OF C THEN RTI;
0263;  " MULTIPLE BYTE COMPARISONS";
0264;  IF B[K].J = D THEN RETURN;
0265;  IF B[K].J ^= D THEN
0266;    DO; END;
0267;  ELSE
0268;    DO; END;
0269;  " CONDITION CODES ";
0270;  IF .ZERO THEN GOTO B;
0271;  ;
0272;  "*** IFF-THEN ***";
0273;  ;
0274;  IFF .A = 5 THEN OUT;
0275;  IFF .C THEN OUT;
0276;  IFF + THEN GO TO OUT; "SAME AS NEXT ONE ";
0277;  IFF + THEN OUT;
0278;  IFF - THEN OUT;
0279;  IFF = THEN OUT;
0280;  IFF ^= THEN OUT;
0281;  OUT:  "TARGET FOR IFF";
0282;  IFF .X = 5 THEN JAMIT;
0283;  IFF .Y < 6 THEN JAMIT;
0284;  IFF .A >= 7 THEN JAMIT;
0285;  IFF .X ^= 3 THEN JAMIT;
0286;  JAMIT:
0287;  ;
0288;  IFF .NOT = THEN MIKE;
0289;  IFF .NOT .CARRY THEN MIKE;
0290;  IFF .NOT .OVERFLOW THEN MIKE;
```

```
0291; IFF .NOT .ZERO THEN MIKE;
0292; IFF .NOT .NEGATIVE THEN MIKE;
0293; IFF = THEN MIKE;
0294; IFF .C THEN MIKE;
0295; IFF .O THEN MIKE;
0296; IFF .Z THEN MIKE;
0297; IFF .N THEN MIKE;
0298; ;
0299; " TARGET FOR IFF ";
0300; MIKE:
0301; ;
0302; "*** INC ***";
0303; ;
0304; INC BETA;
0305; BETA + 1;
0306; ;
0307; "*** INCREMENT WORD ***";
0308; INCW BB;
0309; "*** PAGE ***";
0310; " SEE FIRST LINE FOR ONE WITH TITLE";
0311; PAGE ; " NOTE THAT BLANK IS REQUIRED IF NO TITLE";
0312; ;
0313; "*** PULSE ***";
0314; ;
0315; PULSE BIT[1] OF BETA;
0316; PULSE BIT[1] SET OF BETA;
0317; PULSE BITS[ALPHA] CLEAR OF BETA;
0318; ;
0319; "*** ROTATE ***";
0320; ;
0321; ROTATE LEFT Q.4;
0322; ROL Q.4;
0323; ROR R.1;
0324; ROTATE RIGHT R.2;
0325; " CAN USE SUBSCRIPTS ";
0326; ROR B[Q];
0327; ROR B[Q-5];
0328; ROL B[Q-5].3;
0329; " QUANTIFIER MAY BE EXPRESSION ";
0330; ROL B[K+5-J].Q-J+3;
0331; ;
0332; "*** RTI ***";
0333; ;
0334; LABEL: RTI;
0335; RTI;
0336; ;
0337; "*** SET ***";
0338; ;
0339; SET BITS[5] OF D;
0340; SET BIT[158] OF C;
0341; SET BIT[255] OF D;
0342; " CAN ALSO SET ALL CONDITION CODES BUT .OVERFLOW";
0343; SET .CARRY;
0344; SET .DECIMAL; SET .D;
0345; SET       .D;
0346; SET .INTERRUPT;
0347; ;
0348; "*** SHIFT ***";
```

```
0349;  ;
0350;  SHIFT LEFT B,3;
0351;  SHL B,3;
0352;  SHR C,1;
0353;  SHIFT RIGHT C,1;
0354;  SHR BCK+5 .AND 1],M-K+3;
0355;  ;
0356;  "*** STACK ***";
0357;  ;
0358;  STACK .A, .X, .Y;
0359;  STACK BETA, GAMMA;
0360;  STACK WORD GAMMA;
0361;  STACK 5;
0362;  STACK -5;
0363;  STACK $AB;
0364;  ;
0365;  "*** UNSTACK ***";
0366;  ;
0367;  UNSTACK .Y, .X, .A;
0368;  UNSTACK GAMMA, BETA;
0369;  ;
0370;  "*** WAIT ***";
0371;  ;
0372;  WAIT ON BITS[ALPHA] OF BETA;
0373;  WAIT ON ALL BITS[ALPHA] SET OF BETA;
0374;  WAIT ON ANY BITS[ALPHA] CLEAR OF BETA;
0375;  WAIT ON ALL BITS[ALPHA] SET OF BETA;
0376;  WAIT ON ALL BITS[ALPHA] CLEAR OF BETA;
0377;  WAIT ON BITS[5] OF BETA;
0378;  WAIT ON BIT[4] CLEAR OF BETA;
0379;  ;
0380;  "*** WHILE ***";
0381;  ;
0382;  WHILE XX + 1 < YY
0383;  DO;
0384;  END;
0385;  WHILE 2=2
0386;  DO;
0387;  END;
0388;  ;
0389;  DCL L1 WORD,L2 WORD,L3 WORD;
0390;  DCL XX WORD, YY WORD;
0391;  DCL L5 WORD,LAB2,TS;
0392;  "...ADDITIONAL TESTS NOT IN MANUAL ";
0393;  " MISC. EXAMPLES OF ASSM. EQUIVALENTS ";
0394;  .A=BB;
0395;  .X=5;
0396;  .A=@B[.X] ;
0397;  .A=B[.X];
0398;  .Y=$30;
0399;  .A=B[.Y];
0400;  .A=@B[.Y];
0401;  BB[E]=BB[E]+2;
0402;  F[E+B]=F[E+B+1]+3+D;
0403;  ;
0404;  " TAB CHARACTERS ARE ALSO SUPPORTED FOR CLARITY ";
0405;  "   OF LISTING IN PASS 1 ";
0406;  " EITHER 'TAB' OR 'BACKSLASH' ARE SUPPORTED ";
```

```
0407;  ;
0408;  DCL F;
0409;  ;
0410;  * TEST TAB CHARACTER *;
0411;       FOR I=1 TO 99
0412;            DO;
0413;            I=1;
0414;            IF J=3 THEN
0415;                 DO;
0416;                 F=3;
0417;                 JJ=K+3;
0418;                 END;
0419;            K=3;
0420;            END;
0421;  ;
0422;  * NEXT THE SAME THING WITH BACKSLASH *;
0423;  ;
0424;  * TEST ALTERNATE TAB CHARACTER *;
0425;       FOR I=1 TO 99
0426;            DO;
0427;            I=1;
0428;            IF J=3 THEN
0429;                 DO;
0430;                 F=3;
0431;                 JJ=K+3;
0432;                 END;
0433;            K=3;
0434;            END;
0435;       EXIT;
```

CUT ALONG THIS LINE

DOCUMENT REGISTRATION FORM

Please fill out and return this card to automatically receive all updates to your manual.

DOCUMENT NAME:

DOCUMENT NUMBER: **REVISION:**

(Name)

(Company)

(Street Address)

(City) (State) (Zip)

PLACE
STAMP
HERE

Documentation Manager
D727, RC55
Rockwell International
MICROELECTRONIC DEVICES
3310 Miraloma Ave.
Anaheim, CA 92803

ROCKWELL INTERNATIONAL - MICROELECTRONIC DEVICES

REGIONAL SALES OFFICES

HOME OFFICE*
Rockwell International Corp.
Microelectronic Devices
P.O. Box 3669
Anaheim, Ca. 92803
U.S.A.
Phone: (714) 632-0550
TWX: 910-591-1698

* Also Applications Centers

CENTRAL REGION, U.S.A.
Sales Engineering Office
6631 East Washington Street
Indianapolis, Indiana 46219
(317) 353-9253 Attn: Mid-America Mgr.

EASTERN REGION, U.S.A.*
Caroller Office Building
850-870 U.S. Route 1
North Brunswick, New Jersey 08902
Phone: (201) 246-3630

MIDWEST REGION, U.S.A.
1011 E. Touhy Avenue, Suite 245
Des Plaines, IL 60018
Phone: (312) 297-8862

WESTERN REGION, U.S.A.
3310 Miraloma Avenue
P.O. Box 3669
Anaheim, Ca. 92803
Phone: (714) 632-0550

EUROPE
Rockwell International GmbH
Microelectronic Devices
Fraunhoferstrasse 11
D 8033 Munchen Martinsried
Germany
Phone: (089) 859-9575
Telex: 052 2650

FAR EAST
Rockwell International Overseas Corp.
Ichiban-cho Central Building
22-1 Ichiban-cho, Chiyoda-ku
Toyko 102, Japan
Phone: 265-8808
Telex: J22198

YOUR LOCAL REPRESENTATIVE

ROCKWELL INTERNATIONAL - MICROELECTRONIC DEVICES

REGIONAL SALES OFFICES

HOME OFFICE*
Rockwell International Corp.
Microelectronic Devices
P.O. Box 3669
Anaheim, Ca. 92803
U.S.A.
Phone: (714) 632-0950
TWX: 910-591-1698

* Also Applications Centers

CENTRAL REGION, U.S.A.
Contact Robert O. Whitesell & Associates
6691 East Washington Street
Indianapolis, Indiana 46219
(317) 359-9283 Attn: Milt Gamble, Mgr.

EASTERN REGION, U.S.A.*
Caroller Office Building
850-870 U.S. Route 1
North Brunswick, New Jersey 08902
Phone: (201) 246-3830

MIDWEST REGION, U.S.A.
1011 E. Touhy Avenue, Suite 245
Des Plaines, IL 60018
Phone: (312) 297-8867

WESTERN REGION, U.S.A.
3310 Miraloma Avenue
P.O. Box 3669
Anaheim, Ca. 92803
Phone: (714) 632-0950

E JROPE
Rockwell International GmbH
Microelectronic Devices
Fraunhoferstrasse 11
D-8033 Munchen-Martinsried
Germany
Phone: (089) 859-9575
Telex: 0521/2650

FAR EAST
Rockwell International Overseas Corp.
Ichiban-cho Central Building
22-1 Ichiban-cho, Chiyoda-ku
Toyko 102, Japan
Phone: 265-8808
Telex: J22198

YOUR LOCAL REPRESENTATIVE