NCI/IP-76/08

GPPASM -- A PDP8E ASSEMBLER FOR THE GENERAL PICTURE PROCESSOR

NCI/IP Technical Report #16

December 15, 1976

Peter Lemkin, Bruce Shapiro,
Morton Schultz, Lewis Lipkin,
George Carman*

Image Processing
Division of Cancer Biology and Diagnosis
National Cancer Institute
National Institutes of Health
Bethesda, Maryland 20014

*Carman Electronics, Inc.
Corvallis, Oregon 97330

"We here highly resolve . . ."

NCI/IP-76/08

GPPASM - A PDP8E ASSEMBLER FOR THE GENERAL PICTURE PROCESSOR

NCI/IP Technical Report #16

Peter Lemkin, Bruce Shapiro, Mort Schultz
Lewis Lipkin, George Carman

Image Processing Unit
Division of Cancer Biology and Diagnosis
National Cancer Institute
National Institutes of Health
Bethesda, Md. 20014

December 15, 1976

Abstract

GPPASM, an assembler for the General Picture Processor at the National Cancer Institute, will run on a PDP8e. It is intended to assemble RTPP source code produced by the MAINSAIL compiler on the PDP10 as well as micro instructions for the microinstruction control program memory (MCPM) of the GPP. Binary output files may then be loaded into the GPP by the GPPLDR program.

TABLE OF CONTENTS

SECTION 1

Introduction

The General Picture Processor (GPP) is a special-purpose image processing computer, one of the components of the Real Time Picture Processor (RTPP) ([Carm74], [Lem74], [[Lem76a]). This system, now under construction at the National Cancer Institute, will use powerful techniques of image- processing and artificial intelligence on the images of objects being examined with an automated microscope under control of the RTPP, to provide "intelligent" assistance to the biologist using its facilities.

The system is described more extensively elsewhere [Lem76a]. For the purposes of this document, its important characteristics are these:

1) The GPP component is a processor in its own right. It is not autonomous, but always operates under control of the PDP8e, and has no peripheral devices of its own except buffer memories, and some switches and lights at a control desk. It is optimized for processing images.

2) Part of the network constituting the system is a large, interactive time-shared computer, the PDP-KI10, which will be used to support the RTPP. Extensive software support already exists on the PDP10 for the creation of software.

For these reasons software for the GPP will be created on the PDP10 using the MAINSAIL cross-compiler [Wil75]. GPP source code generated by MAINSAIL on the PDP10 will be transmitted from the PDP10 to the PDP8e, where it will be assembled by the absolute code assembler GPPASM. The load file produced may be loaded by the PDP8e using the absolute binary loader program GPPLDR in the DDTG system [Lem76b] and run on the GPP.

A runtime program for the GPP running on the PDP8e computer may be chained to by GPPLDR in order to provide the GPP program associated with it with a specific runtime system.

This document describes the GPP assembler, GPPASM, which runs on the PDP8e to produce programs which run on the GPP.

It should be noted that GPPASM is a reentrant assembler. Thus a GPPASM program (in GPPMODE) consists of two separate processor load segments (an instruction memory (PM) segment and a data memory (GR) segment). Together with a REQUIRE <PDP8e ".SV" file> RUNTIME specification in the file, this constitutes the loader file for the GPPLDR. This resulting load file is described subsequently.

Alternatively, GPPASM may be used to assemble microinstructions for the microinstruction control program memory (MCPM) and mapping memory (MM) when in MICROMODE. These memories and the microprogram control structure is discussed in portions of this document and in [Lem76a]. The MCPM microcode constitutes the implementation of the GPP "macro" instruction set (eg. ADD, MOVE, etc.).

The mapping memory is used to map the operator values of the macro instruction set (eg. ADD=000120) to corresponding starting addresses of a set of microinstructions in the MCPM. Both the microinstructions and mapping memory source language may be assembled by GPPASM (while in MICROMODE) and loaded by GPPLDR. The default assemble mode is GPPMODE.

It is possible to shift between macro and micro assembler modes defining new instructions in the microassembler mode and using them in the macroassembler mode.

Section 2 discusses the GPP assembly model while Section 3 the microinstruction assembly mode. Section 4 discusses the assembler in general. Sections 5 and 6 discuss running GPPASM and GPPLDR respectively. Section 8 presents the BNF grammar for GPPASM. Section 9 discusses the load file format.

SECTION 2

GPPMODE Assembler Syntax

The syntax of the GPP assembly language (GPPASM) is similar to that of most assembly languages. GPPASM is the assembler for the GPP, and certain of its features are chosen with rapid assembly and ease of documentation in mind. For example: statements are fixed fields and are delimited by carriage returns; comments are delimited by special comment delimiters ("...") in the same way as in SAIL. The full BNF specification is given in the Appendix.

The syntax is roughly as follows:

<label> <PM-statement>

or

<label> <GR-statement>

where <label> is optional.

The <PM-instr.> consists of a fixed format triple operand instruction:

<Operand> <Sourceoperand 1>, <Sourceoperand 2>, <Sinkoperand>

The <sinkoperand> is refered to as P3 effective address and the <sourceoperands> as the P1 and/or P2 effective addresses.

2.1 Labels

A label is any valid non-reserved symbol, terminated (without an intervening space) by a colon (:). More than one label may prefix a line. Examples:

```
        JUMPSPOT:       TARGET:         BULLSEYE:
        PLACE2:
```

And the usual examples of things which aren't legal labels:

```
        2PLACE:         :
        PLACE2 :        :TARGET
```

Labels are not declared beforehand.

2.2 GPPASM operators

Operators are symbols which represent GPP operation codes, GR memory addresses, or assembler operations. Examples:

```
HALT     (assemble here a GPP HALT instruction)
ADD      (assemble here a GPP ADD instruction)
GRBLOCK  (allocate storage in general registers)
```

2.3 GPP Operands

There are two kinds of operands associated with GPP operation codes: a <sinkoperand> is always the P3 field of an operation; <sourceoperands> are the P1 and P2 fields. Neither operand field is required, omitted fields must always be specified by delimiters or be filled with zeros. For example:

```
MOVE A,,B
MOVE A,0,B
ADD  A,B,C
ADD  A B C
```

Some examples of illegal operand specifications are:

```
MOVE A,C
ADD  A,B,C,
```

2.4 Comments

A comment is introduced by the reserved upper-case word "comment" and is terminated by a semicolon (;). For example:

```
Comment          This is a comment
                 and this is the 2nd line of the comment;
```

A comment in the form of a string of characters enclosed in quotation marks (") may appear anywhere. Any character except (") may appear within such a comment. For example:

```
"No quotation marks here"
```

The following is illegal however,

```
"An unmatched " can cause a lot of trouble."
```

2.5 Declarations

General register memory (GR) scalar and array variables are declared with the constructions.

```
GRBLOCK <sizeofarray>
```

Space is allocated for the arrays from the bottom of the GR space on up starting at the last GRORIGIN.

Preloaded GR arrays are specified by

```
GRBLOCK 0 <preload>

<preload>::= <list-of-values> | \\<text>\\
<list-of-values>::= <list-of-values> , <value> |<value>
        |  [ <repeat-times> <delim.> <list-of-values> ]
<repeat-times>::= <value>
<text>::= text string containing no \\, ", or ;
```

SECTION 3

MICROMODE Assembler Syntax

The microassembler syntax is similar to that of the GPP mode assembler. It is invoked by the MICROMODE pseudop.

The microcontrol part of the GPP consists of the microcontrol program memory (MCPM), the mapping memory (MM), the Mreg and the Oreg (the latter two are special fields in the microinstruction). These memories are discussed in more detail in [Carm77].

In general a <macroinst.def.> is an ordered list of <microinstruction>s.

```
<macroinst.def.>::= <macroinst.def.> <microinstruction> |
                    <microinstruction> crlf

<microinstruction>::= / <MCPM-statement> \ |
                    <MCPM-statement> |
                    <MCPM-statement> crlf

<MCPM-statement>::= <MCPM-instr.> |
                    MCPMDEF ID = <value> |
                    MCPMORIGIN <value> |
                    EPSILON

<MCPM-instr.>::= <MCPM-instr.> <delim> <MCPM-opcode> |
                    <MCPM-opcode>


<MCPM-opcode>::= <Mreg> | <Oreg> | <MCPM-ALUs> | <MCPM-bits>
<Mreg>::= [ <16-bit value> ]
<Oreg>::= ( <7-bit value> )
<MCPM-ALUs>::= AL1$ <ALU-value> | AL2$ <ALU-value>
<ALU-value>::= 00| 01 | 02 | 03 | 04 | 05 |06 | 07 | 10 | 11 |
                    12 | 13 | 14 | 15 | 16 | 17
<MCPM-bits>::= P1>A1 | P2>A1 | P3>A1 | CA1>A1 | ALA>A1 | etc.
```

Expression evaluation is only allowed inside of Mreg [...], and Oreg (...) brackets. Labels are defined inside or outside of microinstructions by terminating a symbol with a ":".

A macroinstruction (eg. ADD) is an ordered list of microinstructions. A microinstruction is an unordered list (because they are executed in parallel) of <micro-opcodes> and/or the [Mreg] and/or (Oreg) fields embedded between "/...\". For example:

```
 DOIT:  / MR>A2, [DMAIO], READA2\
or
        / DOIT: READA2, MR>A2, [DMAIO]\
```

or

```
/ MR>A2 DOIT: [DMAIO] READA2\
```

or

```
/ MR>A2
  [DMAIO]
  READA2\.
```

The "OPRMAP <GPP-opcode-value>=<microcode-label>" pseudop maps GPP instruction values (eg. ADD=000120) to the microinstruction sequences so that the link is made between micro and GPP code. This mapping is done in the mapping memory (MM) which for GPP instruction values a and micro instruction labels b the mapping is:

contents(a) <==b.

SECTION 4

GPPASM Assembler Operations

Various pseudo-operations control the assembler's actions, such as the location of instructions and data in memory, assembler mode, the inclusion of source code from a named file, number radix etc.

4.1 Assembly location

The locations at which GPP instructions are assembled into PM, data into the GR, and microinstructions in the MCPM may be controlled with the PMORIGIN, GRORIGIN, and MCPMORIGIN operations. Their use is as follows:

```
PMORIGIN       <location>
GRORIGIN       <location>
MCPMORIGIN     <location>
```

where <location> is an expression evaluated to a 16-bit value designating a specific (non-relccatable) address in the relevant memory space.

The code or data following the xxORIGIN is assembled into the location (modulo 65K for the PM or GR and modulo 8K for the MCPM since this is the size of the memories).

4.2 Expunging symbols

Symbols may be expunged from the permanent symbol table by using the EXPUNGE pseudop as:

```
EXPUNGE ID
```

or a class or symbols may be expunged by mentioning the class type

```
EXPUNGE <class type>
```

where <class type>::= 1 | 2 | 3

where PM=1, GR=2, and MCPM=3.

4.3 Nested source time files

The REQUIRE operation may be used to insert a named source file at an address in a program being assembled. It is used as follows:

```
REQUIRE <filespecification> SOURCETIME
```

where the <file> may be any legal name of a PDP8e file. If the file does not exist, or the <file> is omitted, an error indication is returned; otherwise the file is inserted at this point as if it occurred here. Such insertions may be nested to 16 levels.

4.4 Requiring load time files

It is also possible to require load files to be loaded at load time by specifying an already assembled file as a load module.

```
REQUIRE <filespecification> LOADTIME
```

GPP runtime procedures (such as the MAINSAIL runtimes) may be declared using the LOADTIME REQUIRE statement.

4.5 Runtime file

The runtime environment of the GPP may be specified in the GPP source file or at GPPLDR time. The file is run by being started via the OS8 chain feature from GPPLDR after all loader data is processed (i.e. loaded in the GPP). It may be specified during GPPASM assembly by

```
REQUIRE <PDP8e ".SV" file> RUNTIME
```

4.6 New symbol definitions

GR addresses may be explicitly defined using the GRDEF operator. For example:

```
GRDEF cat = 1234;
GRDEF dog = cat+10;
```

PM operators (such as ADD etc.) may be defined using the PMDEF operator having 20-bit values coded as follows:

```
(opcode group base)*'10000 +
(p1p2p3 use bits)*'100000 +
(ALU number).
```

For example, the ADD instruction uses all p1,p2,p3 fields; is in group 3 with OPR base code 120 and uses ALU 004. Then it would be defined as:

```
        PMDEF ADD = | 3 | 120 | 004
or
        PMDEF ADD = 3120004
```

MCPM opcodes are used in register transfer operations in the GPP microcontroller. They are generated by a bit being on in the appropriate position in the 128-bit MC command register which is loaded from the MCPM. MCPM opcodes, <MCPM-opcode>, (such as READA1, MR>A2 etc.) may be defined using the MCPMDEF operator. The value associated with the symbol is the number of bits to shift the value 1 (where the number of shifts < 128). For example,

```
        MCPMDEF P1>AB1 = 000
        MCPMDEF P2>AB1 = 001
        MCPMDEF P3>AB1 = 002.
```

macroinst.def.s such as ADD, MOVE etc. need to be mapped to the actual microprogram starting addresses. This is done by the OPRMAP pseudop (discussed in Section 3 in more detail). The OPRMAP does NOT add the definition to the symbol table, but rather generates information for the loader file.

```
<MM-statement>::= OPRMAP <GPP-opcode-value> = <MCPM-label>
<GPP-opcode-value>::= <value (i.e. base value+instance value)>
```

4.7 SECTION statement

It is sometimes desirable to label various sections of the code such that a non-executable marker gets passed to the loader. The loader might use this for example to search for sections of a file to load, or it might be used to indicate what sections are in the file. The syntax is:

```
        SECTION <file specification>
```

which puts the specification into the load file at that point.

4.8 GPP starting address

The starting address of the GPP program may be specified either by using the pseudop GPPSTART or through the GPPLDR. For example,

```
        GPPSTART <starting address value in PM>.
```

SECTION 5

Running GPPASM

Running under OS8 (including BATCH), up to nine OS8 partial input files may be assembled as one complete GPPASM source file. The binary, listing and symtab map files are optional and their absence causes that part of the assembly to be aborted. The default extensions for the GPP source and binary files are ".GS" and ".GB" respectively. The extensions for the listing file is ".LS" and for the map is ".MP". The program may be run as follows:

```
.R GPPASM
*<.GB>,<.LS>,<.MP>_<f1.GS>,<f2.GS>,...,<f9.GS>
```

Various switches may also be included:

/D - Debug mode switch which prints out the parse and interpreter stacks.

/G - Load and go (start the GPP) by chaining to GPPLDR and loading the <.GB> file then starting the GPP.

/N - Debug mode switch to parse the input but not to interpret it.

/S - append the symbol table map in GPPLDR readable form at the end of the .GB file.

SECTION 6

Running GPPLDR

The GPPLDR is a separate program which may be run either from DDTG or from OS8 (the latter includes running it under BATCH). Up to nine input files may be specified on each command line. An optional loader map file may be specified on the output command line. The loader is run as follows:

.R GPPLDR
*<Opt. SEC. name>,<Opt. .MP>_<f1.GB>,<f2.GB>,...,<F9.GB>

Various switches may be included to modify the loaded file.

*NEWRUNTIME.SV/N< - specify a new PDP8e runtime file.

*=nnnnn/A - specify a new GPP starting address (up to 32K)

*/G - start the GPP.

*/R - chain to the current PDP8e runtime if it exists.

*<f1.GB>,<f2.GB>,...,<F9.GB>/S - add symbols at the end of the files (created with /S switch in GPPASM) to the DDTG symbol table.

*<f1.GB>,<f2.GB>,...,<F9.GB>/D - delete symbols at the end of the files (created with /S switch in GPPASM) from the DDTG symbol table.

*/X=n - expunge type n symbols from DDTG symtab. n=1 (PM labels), n=2 (GR labels), n=3 (MCPM labels).

*<f1.GB>,<f2.GB>,...,<F9.GB>/P - list the sections contained in the input file list.

*<f1.BG>,<section-name-spec>/L - search <f1.GB> for the start of <section-name-spec.>. Load the data in the section up to the end of data or start of a new section.

SECTION 7

Descriptions of GPPASM Modules

This section contains brief functional descriptions of the modules of GPPASM, the assembler for the GPP. The descriptions indicate the methods of operation and general flow of control.

```
                                MAIN
                               ------
                                 *
                                 *
                                 *
                                 *
                        COMMAND DECODER <----Command Line
                        ---------------
                                 *
                                 *
                                 *
                                 *
SYMBOL TABLE <----          SCANNER      <-------     SEMANTIC ROUTINES
ROUTINES      ----->        -------      ------->     -----------------
                                                             *
                                                             *
                                                             *
                                                             *
                                           Polish String     *
                                                             *
                                                             *
                        Loader Code    <----          CODE GENERATOR
                        Assembly listing <-----       --------------
```

Figure 1. Flow of Control for GPPASM Assembler

7.1 Main program description

Controls initialization of operating variables, invokes input/output initialization, controls passes (first pass plus symbol cleanup). Contains main commentary on program operation and usage. Contains interfaces with the operating system and invoking programs.

7.2 Command decoder description

On being invoked by the main program, the command decoder checks for the presence of a set of file name specifications and switches. It then deciphers the input to determine what files are to be used source code, binary, and listing (if so desired), and what options are to be used in assembly (eg. optional listing file). If there are errors it so informs the user, and requests elucidation at the terminal.

7.3 Input/output description

Opens and maintains files. Provides simple I/O interfaces for file usage to keep details of I/O out of main program logic. Interprets error returns and presents error messages when desirable, and handles I/O errors to whatever extent possible.

7.4 Scanner description

The scanner is a finite state acceptor hand coded produced from the BNF grammar for the GPPASM. The scanner picks from the source stream (source file) individual syntactically significant symbols, encoding them into a form easier for subsequent routines to handle (the symbol table indices which are integer numbers < 2047). It detects and handles syntactic errors at this level.

7.5 Scanner - symbol routines description

The symbol table procedure SYMTAB maintains tables of symbols, their values and types (permanent symbols and created symbols) in a form allowing rapid retrieval of their values and characteristics. Symbols are created by being entered in the appropriate tables, and duplicate symbols are detected and handled (by being rejected or modified).

7.6 Code generators description

Translates a syntactically proper line from GPP or MICROMODE assembly language source code to absolute binary loader input files. Assembler actions (pseudo-operations) are also handled here.

7.7 GR allocator description

Handles the usage of GR memory through GR origin definition and subsequent allocation.

7.8 GR allocator - Space checker description

Checks whether there is space in the GR for the GR data. The operation symbol may cause the allocation of more than one GR location.

7.9 PM allocator description

Handles the usage of PM memory through PM origin definition and subsequent allocation.

7.10 PM allocator - Space checker description

Checks whether there is space in the PM for the PM data.

7.11 MCPM allocator description

Handles the usage of MCPM memory through MCPM origin definition and subsequent allocation.

7.12 MCPM allocator - Space checker description

Checks whether there is space in the MCPM for the MCPM data.

7.13 Symbol cleanup description

Flags undefined symbols and handles them. Deletes unwanted symbols from the symbol tables.

7.14 Listing generator description

If the an output .LS file is specified, it generates an assembly listing file.

7.15 Symbol table map generator description

If the an output .MP file is specified, it generates an assembly symbol table map file.

SECTION 8

GPPASM BNF Grammar Specification

The BNF grammar specification is given for the GPPASM assembler to be used to assemble RTPP programs. Note that MAINSAIL will generate GPPASM assembly language output. The MICRCMODE source programs on the other hand are coded manually.

Note that ID is any identifier which is not a keyword in the grammar, INT is any integer, and <text> is any text not including the symbols \\, ", or :. EPSILON is the null string.

Various terminal symbols whose meaning is not apparent are defined (including the semantics) in [Lem76a].

```
<program>::= <GPPsegment> ^Z

<GPPsegment>::= <GPPsegment> <statement> | <statement> |
                EPSILON

<statement>::= <compiler-mode-statement> crlf |
                <section-statement> crlf |
                <expunge-statement> crlf |
                <number-mode> crlf |
                <comment> |
                <require-statement> crlf |
                <PM-label> <PM-statement> crlf |
                <GR-label> <GR-statement> crlf |
                <MCPM-label> <microinstruction> |
                <MM-statement>

<compiler-mode-statement>::= GPPMODE | MICROMODE

<section-statement>::= SECTION <file>

<expunge-statement>::= EXPUNGE ID | EXPUNGE <class type>
<class type>::= <PM-class> | <GR-class> | <MCPM-class>
<PM-class>::= 1
<GR-class>::= 2
<MCPM-class>::= 3

<number-mode>::= DECIMAL | OCTAL

<require-statement>::= REQUIRE <file> SOURCETIME |
                REQUIRE <file> LOADTIME |
                REQUIRE <file> RUNTIME

<PM-statement>::= <PM-instr.> |
                PMDEF ID = <value> |
                PMORIGIN <value> |
                GPPSTART <PM-label> |
                EPSILON
```

```
<PM-instr.>::= <GPP-opcode> <P1> <delim> <P2> <delim> <P3>

<GR-statement>::= GRBLOCK <GR-list>
                  GRDEF ID = <value> crlf |
                  GRORIGIN <value> crlf |
                  EPSILON

<macroinst.def.>::= <macroinst.def.> <microinstruction> |
                  <microinstruction> crlf

<microinstruction>::= / <MCPM-statement> \ |
                  <MCPM-statement> |
                  <MCPM-statement> crlf
<MCPM-statement>::= <MCPM-instr.> |
                  MCPMDEF ID = <value> |
                  MCPMORIGIN <value> |
                  EPSILON

<MCPM-instr.>::= <MCPM-instr.> <delim> <MCPM-opcode> |
                  <MCPM-opcode>

<MM-statement>::= OPRMAP <GPP-opcode-value> = <MCPM-label>
<GPP-opcode-value>::= <value (i.e. base value+instance value)>

<file>::= <device> <fname> . <ename>
<device>::=   SYS:  |  DSK:   | DSKB:  | DSKC: | DSKD: | DSKE: |
                DSKF: | DSKG: | DSKH: | DTA0: | DTA1:
<fname>::= ID
<ename>::= ID

<comment>::= Comment text ; | " text "

<P1>::= ' <GR-address> | <GR-address> | # <value>
<P2>::= ' <GR-address> | <GR-address> | # <value> | <I/O list>
<P3>::= ' <GR-address> | <GR-address> | <PM-address>

<value>::= <land> | <value> ! <value1> | <ae>
<value1>::= <ae> | <land>
<land>::= <value> & <value1>

<ae>::= <sae>
<sae>::= <term> | <sae> + <term> | <sae> - <term>
<term>::= <factor> | <term> * <factor> | <term> % <factor>
<factor>::= <primary>
<primary>::= <PM-label> | <GR-label> | <MCPM-label> |
                  ( <value> ) | + <primary> | - <primary>

<PM-label>::= <label> | EPSILON
<GR-label>::= <label> | EPSILON
<MCPM-label>::= <label> | EPSILON
<label>::= ID :

<PM-address>::= <PM-label>

<GPP-opcode>::= MOVE | JUMP | PUSHJ | POPJ | INCB | DECB |  BEQ
               | BGE | BLT | BGT | BLE | BNE | HLT | AND | NAND
               | XOR | IMPLIES | OR | NOR | EQV | MOVE | MOVBIT
```

```
                | MOVBS | SHFTR | SHFTL | ROTR | ROTL | GTST |
                LTST | GEST | LEST | DMOVE | DSWP | ADD | SUB |
                MUL | DIV | ADDST | SUBST | MULST | DADD | DSUB
                | DMUL | INC | DEC | MOVEN | MOVEC | DMOVEC |
                IOCLR | YRST | XRST | XCLKB | XCLK | YCLKB |
                YCLK | LINE | MAKYXADDR | GETI1 | GETI2 | GETI3
                | MAX | MIN | DIVST | ANDST | NANDST | XORST |
                ORST | NORST | EQVST | MOVBL | MOVBH | MOVBSL |
                MOVBSH | MOVBSP | LOP1 | LOP2 | COP1 | COP2 |
                PUTBYTE | CLRBH | CLRBL | ANDB | ORB | BSETBL |
                BSETBH | BGEB | BLEB | BEQB | BGTB | BLTB |
                DNAND | DAND | DOR | DXOR | DSHFTL | DSHFTR |
                DINC | DDEC | DINCB | DDECB | DADDST | DMULST |
                DDIVST | DANDST | DNANDST | DORST | DXORST |
                FADD | FSUB | FMUL | FDIV | FMINUS | FLOAT |
                FLOATD | FIX | FIXD | ASR | ASL

<GR-address>::=  <value> | <GR-I/O-address>


<GR-I/O-address>::= <neighborhood-pixels> | <auto-index> |
                <indirect-BM-addresses> | <TTY-I/O> |
                <byte-pointer> | <control-desk> |
                <status-registers> | <dynamic-address-vectors> |
                <GR-I/O-registers>

<neighborhood-pixels>::=  I10 | I11 | I12 | I13 | I14 | I15 |
                I16 | I17 | I18 | I20 | I21 | I22 | I23 | I24 |
                I25 | I26 | I27 | I28 | I30 | I31 | I32 | I33 |
                I34 | I35 | I36 | I37 | I38 <auto-index>::=  A0D
                | A0 | A0I | A1D | A1 | A1I | A2D | A2 | A2I |
                A3D | A3 | A3I | A4D | A4 | A4I | A5D | A5 | A5I
                | A6D | A6 | A6I | A7D | A7 | A7I

<indirect-BM-addresses>::=  PBM0 | PBM1 | PBM2 | PBM3 | PBM4 |
                PBM5 | PBM6 | PBM7 |

<TTY-I/O>::= KRB | KSTATUS | TLS | TSTATUS

<byte-pointer>::= PPOINT | GPNT1 | GPNT2

<control-desk>::=  SW1 | SW2 | SW3 | SWA | DSPLYA | DSPLYB |
                DSPLYC | KNOB01 | KNOB23 | KNOB45 | KNOB67 |

<status-regists>::= PDLCNT | PDL | FXAR | DRA | DRB | EXAR |
                EEXAR | FDRB | STATUS

<dynamic-address-vectors>::= I1XM | I1X | I1XP | I1Y | I2XM |
                I2X | I2XP | I2Y | I3XM | I3X | I3XP | I3Y

<GR-I/O-registers>::= GIN | GOUT

<I/O list>::= ( <list> ) | <I/O list> ! <I/O symbol> |
                <I/O symbol>

<list>::= <list> ! <I/O symbol> | <list> ! INT | <I/O symbol> |
```

```
              INT

<I/O symbol>::= $I1 | $I2 | $I3 | $XP | $X | $XM | $YP | $Y |
                $YM | $RIGHT |$LEFT | $VERTICAL | $HORIZONTAL |
                $BM0 | $BM1 | $BM2 | $BM3 | $BM4 | $BM5 |
                $BM6 | $BM7 | $DOUBLEBUFFER

<GR-list>::= <GR-allocation-size> | 0 <preload>
<GR-allocation-size>::= INT
<preload>::= <list-of-values> | \\<text>\\
<list-of-values>::= <list-of-values> , <value> | <value> |
              [ <repeat-times> <delim.> <list-of-values> ]
<repeat-times>::= <value>
<text>::= text string containing no \\, ", or ;

<MCPM-opcode>::= <Mreg> | <Oreg> | <MCPM-ALUs> | <MCPM-bits>
<Mreg>::= [ <16-bit value> ]
<Oreq>::= ( <7-bit value> )
<MCPM-ALUs>::= AL1$ <ALU-value> | AL2$ <ALU-value>
<ALU-value>::= 00| 01 | 02 | 03 | 04 | 05 |06 | 07 | 10 | 11 |
                12 | 13 | 14 | 15 | 16 | 17
<MCPM-bits>::=  P1>A1 | P2>A1 | P3>A1 | CA1>A1 | ALA>A1 |
              ALB>A1 | ALC>A1 | PDA>A1 | PA2>A1 | MR>A1 |
              GTC>A1 | P1>A2 | P2>A2 | P3>A2 | CA2>A2 | ALA>A2
              | ALB>A2 | ALC>A2 | PDA>A2 | MR>A2 | A1P>A2 |
              P1>D1 | P2>D1 | P3>D1 | CA1>D1 | ALA>D1 | ALB>D1
              | ALC>D1 | PC>D1 | MR>D1 | P1>D2 | P2>D2 | P3>D2
              | CA2>D2 | ALA>D2 | ALB>D2 | ALC>D2 | PC>D2 |
              MR>D2 | W1HD1T | W1LD1T | W1HD1F | W1LD1F |
              W2HD2T | W2LD2T | W2HD2F | W2LD2F | READA1 |
              READA2 | D1>ALA | D1>ALB | D1>PCT | D1>PCF |
              D2>ALA | D2>ALB | D2>PCT | D2>PCF | ALUSET |
              ALUCLR | ALU10 | ALU11 | ALU12 | ALU13 | ALU20 |
              ALU21 | ALU22 | ALU23 | MR0 | MR1 | MR2 | MR3 |
              MR4 | MR5 | MR6 | MR7 | MR8 | MR9 | MR10 | MR11
              | MR12 | MR13 | MR14 | MR15 | MOP0 | MOP1 | MOP2
              | MOP3 | MOP4 | MOP5 | MOP6 | MOP7 | MOP>OP |
              MR>MC | DECMC | JMPMC0 | JMPT | JMPF | PUSHJT |
              PUSHJF | POPJT | POPJF | INCPDL | DECPDL |
              SAVEPT | SAVEPF | ISALDT | ISALDF | INCGTC |
              INCPCT | INCPCF | MHALT | TEST | SET

<delim>::= , | space | tab
```

SECTION 9

Loader image file format

Programs for the RTPP are written and compiled on the PDP10 using the MAINSAIL cross-compiler. The output of MAILSAIL is GPPASM source code. This is then transmitted to the PDP8e and assembled using the GPPASM assembler. The GPPASM produced absolute binary load file is then loaded by the GPPLDR in the DDTG program.

Alternatively, microinstruction programs using the MICROMODE are written manually and assembled with GPPASM on the PDP8e. These are also loaded with GPPLDR. The loader file format is discussed here.

The 12-bit binary file consists of 2 parts: a data section, and a symbol table section. Data is packed in standard OS8 binary 8-bit mode which packs 3-bytes/2 12-bit words as follows:.

```
        --------------------------------------
word 1: |Byte 3 high 4-bits | byte 1 8-bits|
        --------------------------------------
word 2: |Byte 3 high 4-bits | byte 2 8-bits|
        --------------------------------------
```

The symbol table section is used only with DDTG. It contains symbols which address the corresponding PM, GR, and MCPM memories. The maximum size of the PM and GR memory spaces are 65K each while the MCPM is 8K.

The PDP8e RUNTIME file is set up by the GPPLDR for the RTPP and other hardware. It acts as a mini-monitor for the GPP to post images, acquire images, perform some of the MAINSAIL runtimes related to file I/O, communicate with the PDP10, etc.

```
        -------------------------------------
        | Data section | Symbol table section|
        -------------------------------------
```

9.1 Loader data section

There are four distinct data segments in the data section of the loader file: PM, GR, MCPM, MM (microinstruction mapping memory). These are discussed below. Data is written in a continuous stream of groups of 8-bit bytes in OS8 3/2 packed data mode (compatible with OS/8 device handlers). There is a 16-bit checksum used at the end of the file. All data is written as multiples of 8-bit bytes.

A loader datum consists of a variable number of bytes, the first of which is a data type code. This in turn determines the number of bytes required as well as its function.

Code	Data interpretation
1	2 bytes of PM origin
2	2 bytes of GR origin
3	2 bytes of MCPM origin
4	2 bytes of MM (mapping memory) origin
5	8 bytes of PM data (64-bits)
6	2 bytes of GR data (16-bits)
7	16 bytes of MCPM data (128-bits)
8	2 bytes of MM (mapping memory) data (13-bits)
9	14 bytes of A2 format LOADTIME require file
10	14 bytes of A2 format RUNTIME require file
11	2 bytes of GPP starting address
12	1 byte of GPPASM version number
13	2 bytes of OS8 12-bit (LSB) assembly date
14	14 bytes of A2 SECTION file name
15	2 bytes of checksum
16	0 bytes of data - end of data section.
17	8 bytes of symbol table entry: (NAME[1:3],IVAL[1:2],ITYPE[1]) packed 3/2.
18	0 bytes of data - end of file.
19	1 byte = number of PM words (8 bytes/word) following
20	1 byte = number of GR words (2 bytes/word) following
21	1 byte = number of MCPM words (16 bytes/word) following

SECTION 10

References

Carm74. Carman G, Lemkin P, Lipkin L, Shapiro B, Schultz M, Kaiser P:A real time picture processor for use in biological cell identification - II hardware implementation. J. Hist. Cyto. Vol 22, 1974, 732:740.

Carm77. Carman G, Lemkin P, Schultz M, Lipkin L, Shapiro B:Microcontrol Architecture of the General Picture Processor. NCI/IP TEchnical Report #22. In prep.

Lem74. Lemkin P, Carman G, Lipkin L, Shapiro B, Schultz M, Kaiser P:A real time picture processor for use in biological cell identification - I systems design. J. Hist. Cyto. Vol 22, 1974, 725:731.

Lem76a. Lemkin P, G Carman, L Lipkin, B Shapiro, M Schultz:The Real Time Picture Processor: Description and Specification. NCI/IP Technical Report #7, March, 1976.

Lem76b. Lemkin P:Functional specifications for the RTPP monitor /debugger - DDTG. NCI/IP Technical Report #2, Feb. 1976.

Wil75. Wilcox C:MAINSAIL - MAchine INdependent SAIL. DECUS meeting, Languages in Review Session, 1975.

VanL73. VanLehn K:SAIL User Manual. Stanford Artificial Intelligence Laboratory. Memo AIM-204, July 1973

APPENDIX A

GPPASM implementation

GPPASM is implemented in OS/8 Fortran II using interspersed SABR type coding which allows easy access to hardware registers. The system consists of the GPPASM.FT main program and subroutines called by it in a hierarchical tree structure. This section goes into this structure in more detail.

The assembler uses upt to 3 passes through the input file stream. The first pass is used to define all labels by generating code to force the GPP pseudo PC counters (KPCPTR, KGRPTR, KMCPTR) to be incremented and defined appropriately according to how the corresponding memory data would be generated. Undefined symbols are noted on the teletype at the end of the pass 1.

The second pass is used to generate the binary (.GB) (.DA) output file if it is specified in the command decoder. If it is not, then it goes immediately to pass 3. Pass 3 will generate a assembly listing on the 2nd output device if specified. Otherwise, it terminates assembly. If a 3rd output file (.MP) is specified, the symbol table label map is printed after the 3rd pass.

GPPASM.FT is the line scanner/finite state parser for GPPASM, the GPP assembler. It processes the OS8 input file character input stream by parsing it into a stack (IPSTK) of symbol table indices. The stack is then interpreted. The parser section uses a 128 character jump table as part of its finite state machine (FSM). Lower case letters are mapped to upper case before interpreting. The use of the symbol table is explained in more detail in Appendix A.3.

The builtin symbol definitions and interpreter process numbers are defined for the system on a SYS: file "INIGPP.DA" which is compiled by subroutine INIGPP.FT into a 10K core symbol table and later saved with the GPPASM.SV image on disk with a new starting address.

Characters are first loaded into a line buffer "LINE" with pointer "ITTYP" using internal subroutine "GETLINE".

The GPPASM program works as follows. A command decoder command line is input from the teletype via the OS8 command decoder. The command then sets up a list of input files to be assembled as 1 source data stream. Data is read in line by line into LINE(ITTYP) from the input stream, and converted to symbol table indices (by GPPASM internal subroutine PARSE) which are stored in the reverse Polish push down stack "IPSTK" with pointer "IPTOP". Operator precedence [(*,%) over (+,-) etc.] is performed using a temporary operator stack "IOPSTK" with pointer "IOPTOP". Entries in the stack are all the indices of symbols in the symbol table.

All symbols, numbers, single character pseudos (type I operators), multi-character pseudops (type II operators), (and temporaries created during interpretation) are stored as symbols (See Appendix A.3 for more details on their definition and use). Characters are parsed by a set of finite state machines (FSM) using a jump table "TABLE" consisting of a FSM to service one or more input characters. The character to be parsed is in variable "ICHAR". (Note: it is an onto mapping (in the algebraic sense) since many characters have the same FSM, e.g. all upper case letters have FSM "letter" etc.). Further testing is performed within each FSM when required to take the state of GPPASM into account in the parse.

Furthermore, undefined symbols and numbers are also pushed by the parser and interpreter onto a garbage collection stack (ITMPSTK) which is used to clean up the symbol table at the end of the interpreter phase.

The interpreter

The interpreter (external subroutines GINTRP.FT and GSOPS.FT) interprets the Polish stack which was generated during the parse by "PARSE" in GPPASM.FT. Specific GPPASM features are implemented at this point. "GINTRP.FT" scans "IPSTK(IP)" from IP=IPTOP to 0 looking for ITYPE(index) values which are not type I or II operators.

Type I and II operators (single and multi- character operators) are pushed onto IOPSTK while the search for an operand continues. Note that operator precedence was performed in "PARSE" and already exists here. IP is then decremented until an operand is found at which time the top operator in the IOPSTK is evaluated. Operator processes are responsible for popping the IPSTK.

In GPPMODE, when there are no more operators left in the stack but there are operands in IPSTK, the operands are loaded into the PM assembly register (MOPR,MPP1,MP2,MP3). After the instruction is assembled, it is dumped or listed (depending on 2nd or 3rd pass). Checking is done before dumping or listing to see if the # and ' fields are used correctly as well as to see whether a Pi field was used which should not have been used.

In MICROMODE, on seeing a / the microassembly register MQ0[0:127] is cleared. Operands are ORed into MQ0. The instruction is terminated (and dumped or listed) on seeing a \.

Type I operator processes are located in GINTRP.FT while type II processes are located in GSOPS.FT. Note that all input stream numbers are converted according to the OCTAL or DECIMAL switch mode MODENUMBER. All internal arithmetic, however, is performed in decimal.

Errors are noted by an error typeout (residing in GPPASM.FT) consisting of an error number with the rest of the command being ignored (or the entire command if no backup is required). There are two types of errors: fatal (denoted by negative internal error numbers where control goes to OS/8) and non-fatal (denoted by positive internal error numbers where control goes to GPPASM's get next statement from the input stream). The error numbers are listed in Appendix B.

Processes

The actual processes used to implement the actions of GPPASM use additional GPPASM runtime routines GIO, DPCVRT, CODEGEN, CODELIST, GETDEV, OCT, and SYMTAB.

A.1 Logical structure of GPPASM

The logical control structure of the GPPASM parser is outlined below:

```
1. Initialize symbol tables (once only INIGPP.FT).
2. Get TTY command.
  2.1 GETLINE or file ==>line buffer LINE[1:ITTYP].
  2.2 Parse LINE into Polish stack==>IPSTK(IPTOP).
     2.2.1 Define new symbols using SYMTAB.FT.
  2.3 Interpret stack <==IPSTK.
     2.3.1 Process GPPASM functions by calling GINTRP.FT.
```

Logical structure of GINTRP

The logical control structure of GINTRP is outlined below:

```
[1] Initialization of the current IPSTK and IOPSTK
        pointers.
[2] Decrement the current IPSTK pointer IP from IPTOP to 0;
        [2.1] If IPSTK is null then done else goto [4.1];
[3] Look for type I, II processes as
    scan stack.
       [3.1] If one is found, then push it into IOPSTK
             and continue scan.
[4] If the top of the IPSTK is an operand
    then do [4.1] else do [2];
        [4.1] Assemble GPP or MICROMODE instruction;
                Dispatch a type I or II process on
                top of IOPSTK.
[5] Goto [2].
```

A.2 Use of symbols in GPPASM

All operators and data are encoded internally as symbols and the "indices" of these symbols are manipulated. subroutine "SYMTAB" allows the creation, accessing, and modification of symbols. A symbol in the symbol table is a triple (name, value, type). A symbol (NAME[1:3]) is 6 characters or less (left justified, right filled with 0's) in length (6-bit Ascii). It has two associated fields: a value field (IVAL[1:2]) and a type field (ITYPE[1]). These are specified below. The symbol table is initially compiled from the Ascii SYS:INIGPP.DA file. GPPASM common area (GPPCMN.FT) is also initialized during the compilation of the symbol table and is saved in SVGPP.DA being loaded on GPPASM entry and saved on GPPASM exit.

The symbol table in GPPASM can hold up to 1823 (a prime number) decimal symbols of up to 6 characters each. A folded hashing scheme is used on a prime number hash table which is searched modulo 1823 to handle clashes as I<==(I+HASH(X))MOD 1823. Six PDP8e words are used to store the symbol table entry.

A.2.1 The symbol table ITYPE field

The "ITYPE" field of all symbols on stack "IPSTK" are typed either by subroutine INIGPP or the parser "parse" as:

3 for MCPM signal
2 for GPP GR address
1 for GPP PM opr device code
0 for undefined symbols
-1 for GPPASM operators
-2 for GPPASM numbers and switches
-3 for GPPASM multicharacter operators and switches
-4 **not used**
-5 **not used**
-6 **not used***
-7 for GPPASM OS/8 device names (4 char max)
-8 for GPPASM for OS/8 file names (6 characters)
-9 for GPPASM OS/8 file extensions (2 chars)
-10 for PM labels
-11 for GR labels
-12 for MCPM labels
-13 for all defined labels on first pass

A.2.2 The symbol table IVAL[1:2] field

The "IVAL" field is used differently for the different types of symbols.

```
ITYPE             IVAL[1:2]
-----             ---------
ITYPE = 3, IVAL[1] is the # bits to shift '1 for the
                  MCPM signal.
ITYPE = 2, IVAL[1:2] is the GPP GR address
ITYPE = 1, PM instruction:
             IVAL(2)[0:2] - P1P2P2 use bits;
             IVAL(2)[3:11] - OPR group code base;
             IVAL(1)[4:11] - ALU number.
ITYPE = 0, IVAL is information for undefined symbols.
ITYPE = -1, IVAL[1] is the operator precedence
            0 is highest, +n is lowest;
            IVAL[2] bits [6:11] is a type I process pointer
ITYPE = -2, IVAL[1] is MSW, IVAL[2] is LSW of 2's
                  complement number. (Note: NAME[1:3]
                  stores ("!!"&IVAL[1:2]).
ITYPE = -3, is used for three types of special
                  operators. The IVAL[2] field specifies
                  the type II process pointer.
                  IVAL[1] function
                  ------- --------
                  0       type II opr, process # in IVAL[2]
                  1       switch, switch # in IVAL[2]
ITYPE = -7, IVAL[1] is the OS/8 device number
ITYPE = -8, IVAL[1:2] is not used.
ITYPE = -9, IVAL[1:2] is not used.
ITYPE = -10, IVAL[1:2] is the PM address of the label
ITYPE = -11, IVAL[1:2] is the GR address of the label
ITYPE = -12, IVAL[1:2] is the MCPM address of the label
ITYPE = -13, IVAL[1:2] is not used.
```

A.2.3 Label parsing

When a colon is encountered in an input line, the Finite-State-Machine "Colon" is called to parse the label. The initial action is to test the preceeding symbol to be sure it is not null (if it is null, error condition #15 "undefined symbol", is raised).

If a valid symbol preceeds the colon, a symbol table look-up for the symbol occurs. If the symbol did not previously exist in the table, this has the effect of entering the symbol with ITYPE and IVAL[1:2] set to zero. If the symbol did exist, ITYPE (and IVAL) will be returned with their respective values.

ITYPE is then tested. If it is zero, it is set to -13 (IVAL may also be set here) by a second call to the SYMTAB routine. This identfies the symbol as a defined label for the first pass. This value will later be changed to -10, -11 or -12 depending on whether it is a PM, GR, or MCPM label respectively.

If ITYPE = -7, the symbol is an OS/8 device name. An inquiry is then made of the current OS/8 system to determine if the device is active (ie exists) on the system. If it is not active, error condition #5 ("illegal device name") is raised. If it is active, IVAL[1] is set to the internal OS/8 device number and the necessary switches are set to expect a standard <file> arguement following the colon.

If ITYPE is non-zero and not -7, then the symbol was previously defined and error condition #16 ("multiple symbol definition") is raised.

A.3 Internal subroutines

The following subsections (4.5.-) list the internal and external subroutines embedded in the Fortran source subroutines. Externally callable (Fortran II/SABR) subroutines embedded in these sources are marked with "*EX*".

A.3.1 Internal GPPASM subroutines

```
1. PARSE         - Parse input line into Polish stack "IPSTK"
2. GETLINE       - *EX* Get next input line intop line buffer,
                   Break for (carriage return, line feed,
                         =, /, >, <, Ctrl/C)
                   Edit with (rubout, Ctrl/U, Ctrl/T, Ctrl/R,
                   Ctrl/E, Ctrl/Z).
  2.1 AGETLINE   - alternate entry to GETLINE without (CRLF*) on entry.
  2.2 CTLC       - Control/C service, save state and exit.
3. INCHAR        - *EX* Get next input character from input stream.
4. OUT           - *EX*  Print  next character in the output stream.
5. PUSHP         - Push index(CURSYM,IVAL,ITYPE)==>IPSTK[IPTOP].
6. PUSHOP        - Push index(CURSYM,IVAL,ITYPE)==>IOPSTK[IOPTOP].
7. OPMOVP        - Empty IOPSTK==>IPSTK (copy indices).
###### S t a r t     of  finite state machines #########
8. SYMBOL        - FSM: assemble symbol in "CURSYM".
9. LOWER         - FSM: convert lower case to upper.
10. NUMBER       - FSM: assemble number into "CURSYM".
11. COMMA        - FSM: push argument symbol and test ignore THEN.
12. OPERATOR     - FSM: push opr ==>IPSTK (except +,-,%,*).
13. ARITHOP      - FSM: implement operator precedence
                        using IOPSTK and IPSTK.
14. Comment      - FSM: ignore input ICHARi's until next ".
15. PERIOD       - FSM: to process current ptr's or file.ext syntax.
16. COLON        - FSM: to "inquire" from the USR (OS/8) the
                   device name of CURSYM and push device number.
17. FTEXT        - FSM: to process \\...\\
18. OPENMC       - FMS: to process / for opening microinstruction.
19. CLOSEMC      - FMS: to process \ for closing microinstruction.
20. ERROR        - *EX* error routine which either restarts on
                        next line or exits GPPASM.
```

A.3.2 Internal GINTRP subroutines

```
1. CTROTST       - *EX*  test for control/o, terminate interp.
2. BUMPL         - test IP > 1, get symbol at
                   The top of IPSTK, decr. IP.
3. CVIVAL        - *EX*  convert IVAL[1:2] to F.P. # fc
                   based on "MODEN" switch.
4. BINARG        - *EX*  test if binary arguments exist, then
                   get 2 top args IVAL[1:2] (of IPSTK)
                   into IA[1:2] and IB[1:2], and
```

		convert IA to FA, IB to FB.
5.	FCSTORE	- *EX* convert FC to IVAL[1:2] and store "number" symbol into IPSTK(IP).
6.	FILESPEC	- *EX* get the device name, device number, file name and extension into COMMON.
7.	DOSYMTAB	- call SYMTAB(CURSYM,IVAL,ITYPE,INDEX,IDOSYMTAB)
8.	PMDATA	- assemble PM instruction
9.	MCPMDATA	- assemble MCPM instruction

A.3.3 Internal GIO subroutines

1.	ISETDEV	- fetches the input device handler
2.	OSETDEV	- fetches the output device handler
3.	R	- read 1 block into IBUF buffer
4.	W	- write 1 block from JBUF buffer
5.	GETC	- get next character from IBUF
6.	PUTC	- put next character into JBUF
7.	INNC	- *EX* external version of GETC
8.	OUTC	- *EX* external version of PUTC, error in AC

A.3.4 Internal SYMTAB subroutines

1.	SETADR	- set the symbol node ptr from I to NODE.
2.	WBLK	- write the symbol table page back onto the disk.
3.	CBLKOFF	- compute (using EARE) IRELBLK, IRELOFF from index I.

A.3.5 Internal GSOPS subroutines

1.	BUMPL	- test IP > 1, get symbol at the top of IPSTK, decr. IP. with the filename FILE in COMMON

A.4 External FORTRAN subroutine files in GPPASM

The following list of subroutines are used in GPPASM.

1. GIO.FT - General 8 bit Ascii I/O and bloc I/O.
2. SYMTAB.FT - Make, fetch, delete (name,value,type) symbol triples and indices. is for disk simulation version.
3. DPCVRT.FT - D.P. Integer to/from F.P.
4. OCT.FT - D.P. Octal to/from D.P. Integer
5. GINTRP.FT - GPPASM interpreter called after parse is done.
6. GSOPS.FT - function to implement type II interpreter processes
7. CODEGEN - generate code.
7. CODELIST - list assembled code.
9. INIGPP.FT - initialize the symbol table

Subroutine INIGPP.FT is used for Symbol table initialization and returns to OS/8 when it is done.

A.5 Compiling GPPASM and GPPLDR

GPPASM consists of a set of subroutines which may be compiles separately under OS/8 as follows. The set of compile statements is included in the batch program GPPCP.BI running on the PDP8e.

```
.R FORT
*GPPASM.RL,GPPASM.LS_GPPCMN.FT,GPPASM.FT

.R FORT
*GINTRP.RL,GINTRP.LS_GPPCMN.FT,GINTRP.F2

.R FORT
*GSOPS.RL,GSOPS.LS_GPPCMN.FT,GSOPS.F2

.R FORT
*CODELST.RL,CODELST.LS_GPPCMN.FT,CODELST.F2

.R FORT
*CODEGEN.RL,CODEGEN.LS_GPPCMN.FT,CODEGEN.F2

.R FORT
*GIO.RL,GIO.LS_GIO.FT

.R FORT
*GETDEV.RL,GETDEV.LS_GETDEV.FT

.R FORT
*SYMTAB.RL,SYMTAB.LS_SYMTAB.FT

.R FORT
*INIGPP.RL,INIGPP.LS_GPPCMN.FT,INIGPP.FT

.R FORT
*DPCVRT.RL,DPCVRT.LS_DPCVRT.FT

.R FORT
*OCT.RL,OCT.LS_OCT.FT

.R FORT
*GPPLDR.RL,GPPLDR.LS_GPPCMN.FT,GPPLDR.FT

.R FORT
*GSYMTAB.RL,GSYMTAB.LS_GSYMTAB.FT
```

A.6 Building GPPASM.SV and GPPLDR.SV core images

GPPASM.SV may be built on a PDP8e using the following loader sequence. The OS/8 batch file GPPASM.BI contains this sequence.

```
$JOB GPPASM.BI
```

```
.R LOADER
*GIO/O/I/H
*INIGPP/2
*CDREG/2
*SYMTAB
*OCT
*DPCVRT
*GETDEV
*GINTRP
*GSOPS
*CODEGEN
*CODELIST
*GPPASM
*LIB8/L/U
*/M
*GPPASM.MP/M<$
.SAVE SYS:GPPASM.SV

/ADD SYMBOL TABLE FIELDS 5,6,7 TO CORE CONTROL BLOCKS
.R ABSLDR.SV
*F567.BN=40276$
.SAVE SYS:GPPASM

/BUILD THE SYMBOL TABLE AND SAVE IT
.RUN SYS:GPPASM
.SAVE SYS:GPPASM; 40303

/BUILD GPPLDR.SV IMAGE
.R LOADER
*GIO/O/I/H
*CDREG/2
*GETDEV
*OCT
*DPCVRT
*GSYMTAB
*GPPLDR
*LIB8/L/U
*GPPLDR.MP/M_$
.SAVE SYS:GPPLDR.SV
```

APPENDIX B

List of error numbers

When an error occurs in any procedure in GPPASM, an internal error number is generated and is passed backwards from the error condition to internal subroutine ERROR in GPPASM.FT. The error number is passed through the COMMON variable IERRNUM.

ERROR searches file "SYS:GPPERR.DA" for the error number and prints it and its associated error message. GPPASM then clears various GPPASM switches and restarts at the "*" command level.

ERROR CODE ALLOCATION

```
000:099 - GPPASM ERRORS
100:199 - GINTRP ERRORS
200:299 - GPPLDR ERRORS
200:299 - free
300:399 - free
400:499 - free
500-599 - free
600:699 - free
700:799 - free
800:899 - free
900:999 - free
```

ERROR LIST

```
  0 !!! ILLEGAL ERROR MESSAGE NUMBER !!!
  1 <DIGITS><LETTERS> IS ILLEGAL SYMBOL
  2 UNTERMINATED QUOTE (")
  3 DOLLAR ($) IS NOT FIRST CHAR OF SYMBOL WHERE IT APPEARS.
  4 "FATAL" SPOOLER OUTPUT ERROR.
  5 ILLEGAL OS/8 DEVICE NAME.
  6 ILLEGAL PARSE CHARACTER.
  7 ILLEGAL <FILE> SPECIFICATION.
  8 ILLEGAL UNARY OPERATOR SEQUENCE
  9 TOO MANY DIGITS IN <NUMBER>
 10 ILLEGAL USE OF / IN GPPMODE
 11 SIXBIT CONVERSION ERROR
 12 FATAL: NO SVGPP.DA FILE ON SYS:
 13 **NOT USED**
 14 PARENTHESIS MIS-MATCH ERROR
 15 UNDEF SYMBOL - REMAINDER OF LINE NOT PARSED
 16 MULTIPLE SYMBOL DEFINITION - REMAINDER OF LINE NOT PARSED

101 BINARY OPR DOES NOT HAVE 2 ARGS
102 FATAL: SYMTAB OVERFLOW DURING INTERPRETATION
103 UNARY OPERATOR ARG ERROR
104 ILLEGAL ASSIGNMENT SYNTAB.
105 ILLEGAL FILE NAME SPECIFICATION SYNTAX
106 FILE LOOPUP FAILED (FILE DOES NOT EXIST)
```

107 UNDEF SYMBOL - NO OPERATION PERFORMED WITH COMMAND LINE.
108 **NOT USED***
109 TWO MANY GPP INSTRUCTION OPERANDS
110 TWO FEW GPP INSTRUCTION OPERANDS
111 OPERAND IN OP STACK WITHOUT GPP INSTRUCTION
112 ILLEGAL USE OF # IMMEDIATE FOR P3 FIELD
113 ILLEGAL USE OF ' INDIRECT OPERATOR ON NULL OPERAND
114 GRBLOCK ARG ERROR
115 MREG ARG ERROR
116 OREG ARG ERROR
117 NO [ ] IN GPPMODE UNLESS IN GRBLOCK
118 NO ( ) IN GPPMODE
119 ILLEGAL / IN GPPMODE
120 ILLEGAL \ IN GPPMODE
121 SAW \ BEFORE / IN MICROMODE
122 ILLEGAL CODEGEN OPERATOR
123 ILLEGAL CODELIST OPERATOR
124 GREATER THAN 16 SOURCE REQUIRE FILES
125 REQUIRE STATEMENT -TIME ARGUMENT ERROR
126 GRBLOCK RAN OUT OF DATA FROM IPSTK - COMPILER MALFUNCTION
127 ILLEGAL \\...\\ TEXT SPECIFICATION
128 UNTERMINATED \\...\\
129 ILLEGAL USE OF # OR ' IN GPP INSTRUCTION
130 ILLEGAL USE OF P1, P2 OR P3 IN GPP INSTRUCTION
131 ILLEGAL IVAL(2)[0:2] P1P2P3 USE CODE SPEC.

201 REQUIRE LOAD FILE LOOKUP ERROR
202 REQUIRE LOAD FILES > 16 FILES
203 HANDLER FOR N - PM WORDS NOT IMPLEMENTED
204 HANDLER FOR N - GR WORDS NOT IMPLEMENTED
205 HANDLER FOR N - MCPM WORDS NOT IMPLEMENTED
206 ILLEGAL /X EXPUNGE TYPE (=NNNNN IN COMMAND DECODER)

INDEX