WKBPCHDP Program

1. Program Overview

The `WKBPCHDP` program is a batch utility developed for execution on an IBM mainframe. It is designed to dynamically allocate files, copy records from a Hadoop-extracted dataset, and validate the record count against a corresponding metadata file (typically a `.DONE` file). This ensures data completeness and accuracy before further processing or downstream integration.

---

2. Header and Documentation Block

- Main Procedure: `WKBPCHDP`
- Entry Point: Declared with `OPTIONS(MAIN)`
- Parameter: Accepts a runtime `PARM` string, which includes input dataset name and record length.
- Author / Date: Captured in block comments
- Summary: Explained through inline documentation. Clearly outlines expected inputs (Hadoop data, .DONE meta file) and outputs (validated copy file).

---

3. Macro and Variable Declarations

Macro Variables

Macro declarations use `%DCL` to define constants and `%DEACTIVATE` to prevent unintentional macro substitution elsewhere in the code.

Macro Procedures

- String Padding Macro: Pads a string to the nearest multiple of 71 characters for formatting.
- Debug Initialization Macro: Returns `INIT(value)` if debug mode is active.
- 16-Bit Integer Conversion Macro: Ensures safe assignment and validation of 16-bit binary integer values using explicit masking and bounds checks.

---

4. Utility Procedures

`STRIP` Procedure

A local subroutine used to remove leading and trailing blanks from strings. Implemented in two versions:

- For VARYING strings
- For fixed-length `CHAR(n)` strings

---

5. File Declarations

The following file declarations are made to facilitate I/O operations:

- `SYSPRINT`: Standard output/log file.
- `HADOUT`: Input file containing data extracted from Hadoop.
- `HADDNE`: Input metadata file (typically `.DONE`).
- `FILOUT`: Output file that will receive the copied records from `HADOUT`.

---

6. Record Structure Definitions

Structures are declared for interpreting and managing fixed-length records:

- 84-byte and 168-byte structures: Match known record formats from Hadoop.
- 80-byte format: Used for control and metadata lines (e.g., `.DONE` files).

Each layout is built using `DCL 1`, `5` level field declarations to represent logical fields.

---

7. Dynamic Allocation Structures

`DYNAMAL`

Entry point used to invoke SVC 99 dynamic allocation service for temporary dataset allocation.

`DYNALLOC`

A structured parameter block that holds DSN names, DD names, DISP settings, and other allocation attributes required by `DYNAMAL`.

---

8. Miscellaneous Variable Declarations

- Flags: Used to detect EOF on input files, allocation success/failure, etc.
- Counters: Maintain the number of records read, copied, and validated.
- Parameters: Hold parsed values such as dataset name and logical record length.

---

9. Built-in Function Declarations

Commonly used PL/I built-ins like:

- `ADDR`: Returns address of a variable or structure.
- `PLIRETC`: Sets a return code for the job step.
- `MOD`, `INDEX`: Used for string and numerical manipulation.

---

10. Main Logic Flow

End-of-File Handling

EOF flags are set when either input file reaches its end. This prevents invalid reads and controls loop exits.

Parameter Parsing

The program reads and parses the `PARM` string, extracting the Hadoop file name and logical record length (`RECLEN`), which determines the structure to use for reads.

File Allocation and Error Handling

Dynamic allocation of the Hadoop input (`HADOUT`) and the `.DONE` metadata file (`HADDNE`) is performed using a call to `ALLOCATE_FILE`. Any allocation failure is handled by a call to `DYNAMIC_ALLOC_ERR_ROUTINE`, which prints errors and stops the program.

---

11. Data Copying Phase

Once files are allocated:

- Records are read from `HADOUT`, using the correct layout based on `RECLEN`.
- Each record is written to `FILOUT`.

- A counter is incremented for each record copied.
- At EOF, the record count is printed to `SYSPRINT`.

---

12. Metadata Validation Phase

- The program reads the `.DONE` file (`HADDNE`).
- Extracts the expected record count from it (usually a simple numeric value in the first 80-byte line).
- Compares it to the counter from the copy phase.
- If the counts do not match, an error message is printed, and `PLIRETC(8)` or `PLIRETC(16)` is called.
- If they match, a success message is logged, and the return code is set to zero.

---

13. Subroutines

`ALLOCATE_FILE`

Handles dynamic allocation of input files using system services.

`DYNAMIC_ALLOC_ERR_ROUTINE`

A structured error handler for dynamic allocation failures. Captures the failing DSN/DD and reason code, and prints diagnostics to `SYSPRINT`.

---

14. End of Program

All files are closed, and the program terminates. The return code reflects the success or failure of the record count validation phase.

---

15. Summary

Purpose

The primary purpose of `WKBPCHDP` is to:

- Dynamically allocate Hadoop data and control files.
- Copy all records from the main input file into a validated output dataset.
- Ensure the copied record count matches the expected count from a `.DONE` metadata file.

Execution Flow

1. Parse runtime parameters.
2. Dynamically allocate input datasets.
3. Copy records while counting.
4. Validate record count using `.DONE` file.
5. Print results and return appropriate job code.

There are six procedures in this PL/I program:

1. **WKBPCHDP** – The main procedure (with OPTIONS(MAIN)).
2. **$CHARL71** – Macro procedure to pad a string to a multiple of 71 characters.
3. **DEBUG_INIT** – Macro procedure for debug initialization.
4. **TOBIT16** – Macro procedure to generate code for assigning a value to a 16-bit logical integer.
5. **STRIP** – Generic procedure with two implementations:
    - STRIP_VARYING_LENGTH
    - STRIP_FIXED_LENGTH
6. **ALLOCATE_FILE** – Subroutine to allocate and open Hadoop output files.
7. **DYNAMIC_ALLOC_ERR_ROUTINE** – Subroutine to handle dynamic allocation errors.
   :

1WKBPCHDP: PROC (PARM) OPTIONS(MAIN) REORDER;

```
None
1WKBPCHDP:  PROC (PARM) OPTIONS(MAIN) REORDER;
```

- Defines the main procedure WKBPCHDP, which takes a parameter PARM. OPTIONS(MAIN) means this is the entry point.

Macro Variable Declarations

These %DCL and %DEACTIVATE statements are macro variable declarations used for controlling formatting and behavior in included macros/structures.
For example:

```
None
%DCL ATTR_DICT     CHAR      /* CONTROLS DISPLAY IN PLINC.DCTEDIT   */;
%    ATTR_DICT     = '';
%DEACTIVATE        ATTR_DICT;
```

- Declares a macro variable ATTR_DICT of type CHAR, sets it to an empty string, and then deactivates it (so it won't be substituted in code).

This pattern repeats for several variables, each controlling formatting or structure options for included macros.

Boolean and Utility Macro Variables

```
None
%DCL FALSE         CHAR;
%    FALSE         = ''0''B)';
%DCL TRUE          CHAR;
%    TRUE          = ''1''B)';
%DCL FOREVER       CHAR;
%    FOREVER       = 'WHILE(''1''B)';
```

- Defines macro variables for boolean values and a loop construct.

Space Padding Macro Variables

None

```
%DCL SPACE         CHAR;
%    SPACE         = '('' '')';
%DCL SPACE2        CHAR;
%    SPACE2        = '('''  '')';
...
%DCL SPACE20       CHAR;
%    SPACE20       = '('''                    '')';
```

- Defines macros for strings of 1 to 20 spaces, useful for formatting output.

End-of-Block Macro Variables

None

```
%DCL ENDIF         CHAR;
%    ENDIF         = 'END /* OF THE IF */';
%DCL ENDELSE       CHAR;
%    ENDELSE       = 'END /* OF THE ELSE */';
...
%DCL ENDOTHERWISE  CHAR;
%    ENDOTHERWISE  = 'END /* OF THE OTHERWISE */';
```

- Macros for marking the end of code blocks, mainly for code clarity and documentation.

Macro Procedures

None

```
$CHARL71
%$CHARL71: PROC(TEXT) RETURNS(CHAR);
  DCL TEXT   CHAR;
  DCL RESULT CHAR;
  DCL LENGTH BUILTIN;
  DCL LEN    FIXED;
  DCL PAD    FIXED;
  LEN    = LENGTH(TEXT);            /* LENGTH OF ARGUMENT */
  PAD    = (LEN/71+1)*71-LEN;       /* REQUIRED PADDING */
  RESULT = TEXT;
  DO LEN = 1 TO PAD;
     RESULT = RESULT || ' ';
     END;
  RETURN(RESULT);
%END      $CHARL71;
%ACTIVATE $CHARL71;
```

- Macro procedure that pads the input string TEXT with spaces so its length is a multiple of 71 characters. Useful for formatting output or generated code.

DEBUG_INIT

None

```
%DEBUG_INIT: PROC(VALUE) RETURNS(CHAR);
  DCL VALUE  CHAR;
  DCL RESULT CHAR;
  IF DEBUG ^= 'Y' THEN RETURN('')     /* IF NO DEBUG CODE    */ ;
  RESULT = 'INIT(' || VALUE || ')'    /* BUILD RESULT STRING */ ;
  RETURN(RESULT);
%END      DEBUG_INIT;
%ACTIVATE DEBUG_INIT;
```

- Macro procedure for debug initialization. If the macro variable DEBUG is not 'Y', it returns an empty string; otherwise, it returns a string like INIT(value).

TOBIT16

```
None
```

```
%TOBIT16: PROC(SOURCE, TARGET) RETURNS(CHAR);
  DCL SOURCE CHAR;
  DCL TARGET CHAR;
  DCL WORK   CHAR;
  TOBIT16_DCL = 'Y'                /* INDICATE DCL REQUIRED */;
  WORK =
  $CHARL71(' /* ' || TARGET || ' = ' || SOURCE || ' */')         ||
  $CHARL71('    TOBIT16_WORD = ' || SOURCE || ';')               ||
  $CHARL71('    IF TOBIT16_WORD > 65535 THEN SIGNAL OVERFLOW;')  ||
           '    ' || TARGET || ' = SUBSTR(UNSPEC(TOBIT16_WORD),17,16)';
  RETURN(WORK);
%END        TOBIT16;
%ACTIVATE   TOBIT16;
%DCL        TOBIT16_DCL CHAR      /* FIRST TIME SWITCH */;
%           TOBIT16_DCL = 'N';
%DEACTIVATE TOBIT16_DCL;
```

- Macro procedure to generate code that assigns a value to a 16-bit logical integer.
  - It creates code that checks if the value fits in 16 bits and assigns it, or signals an overflow.
  - Uses $CHARL71 to pad each generated line for formatting.

Summary:
This section sets up macro variables and macro procedures for use throughout the program. These macros help with formatting, code generation, and conditional debugging, making the main logic more readable and maintainable. No actual business logic is performed here—it's all setup for later code.

Inputs
HADOUT.DAT:
The main binary input file (copied from Hadoop).
HADOUT.DAT.DONE:
The meta file (text) that contains the expected record count (look for a line like RECORD COUNT=...).

Processing
Dynamically allocate (in PL/I; in Python, just open) the input files.
Copy each record from HADOUT.DAT to FILOUT.DAT, counting the number of records processed.
Log all steps and errors to SYSPRINT.TXT.

Validation
After copying, read the record count from HADOUT.DAT.DONE.
Compare the count from the .DONE file to the number of records actually copied.
Log whether the counts match (success) or not (failure).

Outputs
FILOUT.DAT:
The output binary file (copy of HADOUT.DAT).
SYSPRINT.TXT:
Log file with process details and validation results.

Overview

The `WKBPCHDP` program is designed to process data files extracted from Hadoop and transferred to a mainframe (or compatible platform). Its primary purpose is to:

- Copy records from a binary input file to an output file.
- Validate that the number of records copied matches the expected count provided in a companion metadata file.

The program logs all activity and results to a system print file (`SYSPRINT.TXT`) for auditability and troubleshooting.

Inputs

HADOUT.DAT

Main binary input file containing fixed-length records extracted from Hadoop. The records are typically 84 or 168 bytes long.

HADOUT.DAT.DONE

Text metadata file containing the expected number of records in the input file.

Example format:

```
None
RECORD COUNT=12345
```

Key Execution Flow

1. Initialization

- Parses the input `PARM` to determine dataset names and record length.
- Initializes counters and control flags.

2. Open Files

- Opens `HADOUT.DAT` for reading (binary input).
- Opens `FILOUT.DAT` for writing (binary output).
- Opens `HADOUT.DAT.DONE` for reading (text input).

3. Copy Records

- Reads each fixed-length record from `HADOUT.DAT`.
- Writes each record to `FILOUT.DAT`.
- Increments a record counter for each successful read/write.

4. Log Record Counts

- Writes the total number of records read and written to `SYSPRINT.TXT`.

5. Validate Record Count

- Parses `HADOUT.DAT.DONE` to extract the expected record count.
- Compares the actual count (from copy step) with the expected count.
- Logs validation results to `SYSPRINT.TXT`.

6. Error Handling

- If any file is missing or any step fails, logs a detailed error message and exits with a non-zero return code.

Outputs

FILOUT.DAT

Binary output file containing a copy of all records from `HADOUT.DAT`.

SYSPRINT.TXT

Text log file containing:

- The number of records read and written.
- Validation results (success or failure).
- Any errors encountered during processing.

Validation Logic

The record count validation ensures data integrity between the Hadoop-extracted input and the metadata control file.

- If the number of records copied matches the count in `HADOUT.DAT.DONE`, the process is successful.
- If there is a mismatch, an error is logged and the return code is set accordingly.

Example Log Output (`SYSPRINT.TXT`)

Success:

```
None
TOTAL HADOUT OUTPUT RECORDS READ.............     12345
TOTAL HADOOP RECORDS COPIED..................     12345
FTP VALIDATION IS SUCCESS..
MF REC COUNT MATCHED WITH META FILE COUNT.
```

Failure:

```
None
FTP VALIDATION FAILED...
MF REC COUNT DID NOT MATCH META FILE COUNT
```

Summary Table

File Name	Type	Purpose
HADOUT.DAT	Input	Main binary data file from Hadoop
HADOUT.DAT.DONE	Input	Metadata file containing expected record count
FILOUT.DAT	Output	Output file containing copied records
SYSPRINT.TXT	Output	Log file with process stats and validation