Contents

TSDSUTIL

Scripting Language User's Guide

Version 1.0.1

Tommy Sprinkle

2026

Copyright and License

Without the prior written permission of Tommy Sprinkle, neither the Software nor any derivative work, nor any source code, object code, documentation, or other copyrighted material from the Software, may be sold, licensed for a fee, bundled or incorporated into, or otherwise supplied as part of a paid product, package, distribution, or offering.

Commercial Use

The restriction on paid distribution does not prohibit commercial use of the Software. In particular, the Software may be used internally by a commercial organization; may be used in the development, testing, operation, maintenance, or support of commercial products; and may be used in providing paid consulting, support, hosting, processing, or other services.

Use of the Software in a software-as-a-service or hosted service is permitted when the Software itself is not the product being sold or licensed and users are not being charged for receiving a copy of the Software.

Commercial products may depend upon or interoperate with the Software. Users, package managers, build systems, deployment systems, or installers may retrieve the Software separately from an independent source. A commercial provider may not itself bundle, redistribute, or supply the Software as part of its paid product, package, distribution, or offering without prior written permission.

No Source-Publication Requirement

This license does not require publication, disclosure, or distribution of source code for modifications or derivative works that are not distributed.

Other Rights

Nothing in this license limits rights that otherwise exist under copyright law, including fair use and other applicable limitations or exceptions.

No patent license is granted or implied by this license.

Violations and Termination

If the Copyright Holder gives written notice of a material violation of this license, the violating party must stop the violation and cure it within 30 days after receipt of the notice. If the violation is not cured within that period, the rights granted by this license to the violating party terminate.

Upon termination, the violating party must stop using and distributing the Software and any modifications or derivative works except to the extent necessary to comply with applicable law or to complete the cure directed by the Copyright Holder.

Requests for permission may be directed to:

`Tommy Sprinkle - tommy@tommysprinkle.com`

Disclaimer of Warranty and Limitation of Liability

THE SOFTWARE IS PROVIDED “AS IS,” WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NON-INFRINGEMENT.

IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE OR THE USE OF THE SOFTWARE.

About This Guide

This guide documents the TSDSUTIL scripting language and user-visible services. It is written for users of TSDSUTIL rather than as an implementation guide.

System/370 machine instructions are generally documented only where TSDSUTIL behavior differs from System/370 behavior. The appendices provide compact reference material for instructions, macros, return codes, diagnostics, storage layout, and complete examples.

This authoritative Markdown guide is synchronized through **TSDSUTIL Version 1.0.1**.

Chapter 1 — Introduction and Getting Started

Applies to: TSDSUTIL Version 1.0.1

1.1 What TSDSUTIL Is

TSDSUTIL is a scripting utility whose language is based on IBM System/370 assembler. A program can define data with assembler-style declarations, execute supported System/370 machine instructions, call TSDSUTIL macros and services, and process Linux files or AWS-format virtual tape data sets.

The language deliberately uses familiar mainframe concepts: labels, general registers, DSECTs, `DC` and `DS`, base-and-displacement addressing, DCBs, OPEN/CLOSE, and GET/PUT.

TSDSUTIL is not a complete System/370 emulator or IBM operating system. It provides a controlled execution environment for utility-style programs. Supported machine instructions normally follow their System/370 definitions; this manual documents TSDSUTIL-specific differences and restrictions where they matter.

Source is assembled before execution. Assembly-time errors are reported before execution begins; conditions that depend on runtime state, such as an invalid storage reference or an OPEN failure, are detected during execution.

1.2 A Brief History of TSDSUTIL

The ideas behind TSDSUTIL date to the late 1970s, when I worked as an application programmer for Electronic Data Systems (EDS), maintaining a banking system written in IBM Assembler. At EDS we regularly used locally developed utilities, including one called WAAPDSUT. My memory of the details is necessarily incomplete after nearly fifty years, but I believe WAAP identified the group responsible for the utility and DSUT probably stood for Data Set Utility.

WAAPDSUT was a general-purpose data-set utility. We used it to read master files, locate and modify selected records, write updated data sets, and produce reports. What made it especially interesting was its control language: statements were modeled after IBM System/360 and System/370 Assembler machine instructions. For programmers already familiar with assembler, it was easy to learn and powerful enough to replace many small one-purpose assembler programs.

I left EDS in 1979 and immediately missed having the utility available. That prompted me to write a small utility of my own that could perform some of the same kinds of tasks. Over the following decades I returned to the idea many times. The implementations varied widely and none was intended as a recreation of WAAPDSUT, but they all shared an assembler-style control language. Unfortunately, none of those early versions survived.

Eventually I moved the idea from IBM Assembler to C. That changed the nature of the project: instead of relying on the host processor's machine instructions, the utility itself had to implement register operations, storage references, condition codes, branches, and other instruction behavior. That became the genesis of **TSDSUTIL - Tommy Sprinkle's Data Set Utility**.

The current TSDSUTIL was started from scratch and has evolved into something closer to a scripting language, combining ideas from IBM Assembler, the System/370 architecture, and C. I sometimes call it **Monti-Python**. Thinking of TSDSUTIL as almost the opposite of Python led to the phrase *Anti-Python*, which in turn brought *Monty Python's Flying Circus* to mind. The spelling became *Monti* to keep the connection with *Anti*.

TSDSUTIL now emulates much of the non-privileged System/370 instruction set of roughly the mid-1970s, excluding floating point, and surrounds that execution environment with MVS-inspired data-management and debugging services. Development began on Linux, with operating-system-dependent code deliberately separated. A long-term goal is to port TSDSUTIL to **MVS 3.8 under Hercules**, an operating system with which I have a long personal history.

Why build another version now? Because I never quite felt that I had finished the project I started almost fifty years ago. After retiring from professional software development, I finally had the opportunity to return to it properly.

I don't know whether anyone else will find TSDSUTIL useful - or even amusing - but I am making it available under the terms in the LICENSE file. Sometimes you have to build something just because you can.

The complete story, including more of the WAAPDSUT background and the development path that led to the current program, is in the root `HISTORY.md` file.

1.3 Typical Uses

TSDSUTIL is particularly suited to record-oriented utility programs:

Use	Typical facilities
Select records and produce reports	DCB, OPEN/GET/CLOSE, DSECTs, `$PRINT`
Modify records and create a new data set	GET/PUT, DSECTs, character/decimal instructions
Describe structured records	DSECT, USING, `$DFMT`
Process fixed or variable records	F/FB/V/VB record formats and variants
Process AWS virtual tape	SL/NL/BLP tape support, GET/PUT
Inspect or position physical tape	`$READ`, `$WRITE`, `$CONTROL`
Build diagnostic/test programs	`$ASSERT`, `$DUMP`, `$DFMT`, TRACE, `$SHOWDCB`
Exercise instruction sequences	Supported System/370 machine instructions

A TSDSUTIL program can also be very small; data sets and elaborate data structures are not required merely to execute a few instructions or services.

1.4 Program Structure

A program normally contains data definitions followed by executable code. `START` begins executable code and `END` terminates the source program.

```
MESSAGE  DC    C'HELLO FROM TSDSUTIL'
*
HELLO    START ,
         $PRINT MESSAGE,L'MESSAGE
         $EOJ   0
         END    ,
```

This example defines EBCDIC data, begins execution at `HELLO`, prints the data, ends execution with completion code zero, and terminates the source with `END`.

Labels begin in column 1. Statements without labels are normally indented so that the operation does not begin in column 1. Chapter 2 defines the source-language rules; Chapter 3 covers data definition and addressing.

1.5 A First Program

A minimal program needs no data storage:

```
HELLO    START ,
         $PRINT "Hello from TSDSUTIL"
         $EOJ   0
         END    ,
```

Save the source as `hello.tsds`. On Linux, the primary source is read from standard input:

```
./tsdsutil <hello.tsds
```

The program writes:

```
Hello from TSDSUTIL
```

and returns a host process status of zero.

To request an assembly listing:

```
./tsdsutil --listing=hello.lst <hello.tsds
```

To assemble and list without executing:

```
./tsdsutil --no-execute --listing=hello.lst <hello.tsds
```

Chapter 12 is the complete Linux command-line reference.

1.6 Assembly, Execution, and Configuration

Unless execution is disabled, TSDSUTIL parses and assembles the entire program and begins execution only after assembly completes successfully. An unresolved symbol can therefore prevent execution, while a problem that depends on runtime state cannot be detected until the affected statement executes.

Configuration can come from system and user configuration files, source `#PRAGMA` statements, and command-line options. Chapter 9 documents configuration and precedence; Chapter 12 documents the command-line interface.

1.7 Completion and Service Return Codes

Three kinds of status should not be confused:

Status	Purpose
Service return code	Reports the result of an individual service, commonly in R15
Program completion code	Application-selected normal completion, normally established by `$EOJ`
ABEND / host status	Reports abnormal termination or a host-visible process result

For example:

```
$EOJ    0
```

ends normally with completion code zero, while:

```
$EOJ    8
```

ends normally with application completion code 8. On Linux, a normally executed program returns the low eight bits of the TSDSUTIL completion code as the process exit status.

A recoverable service return in R15 does not automatically become the final program completion code; the program can inspect it and decide what to do next. Appendix D is the authoritative return-code and ABEND reference.

1.8 Listings, Diagnostics, and Runtime Output

TSDSUTIL separates several useful forms of output:

Output	Purpose	Primary reference
Assembly listing	Source, addresses/object information, diagnostics, optional XREF and execution information	Chapter 11
Diagnostics	Assembly/runtime errors, warnings, severe and informational conditions	Chapter 10 and Appendix E
Runtime messages	Program output from services such as `$PRINT`, `$DUMP`, `$REGS`, and WTO	Chapter 5
TRACE	Detailed execution trace	Chapter 10
PERFORMANCE	Optional detailed execution profile	Chapter 10

Listing line numbers are used by diagnostics and remain useful when `#COPY` or `#INCLUDE` brings statements from multiple source files into the assembled program.

1.9 EBCDIC and Host Data

Program character data is EBCDIC. For example:

```
NAME     DC    C'MICKEY'
```

stores EBCDIC bytes in TSDSUTIL virtual storage. Machine instructions therefore see the byte representation expected by an IBM assembler program.

Host-facing services perform conversion where required; for example, `$PRINT` can display EBCDIC program data as normal Linux terminal text. TSDSUTIL does not, however, imply automatic translation of arbitrary data-set records. Chapters 6 and 7 describe host-file and AWS-tape processing behavior.

1.10 Where to Go Next

The chapters are organized by purpose:

Need	Continue with
Source syntax, symbols, expressions, literals	Chapter 2
Data definition, DSECTs, registers, addressing	Chapter 3
System/370 machine instructions	Chapter 4
TSDSUTIL macros and services	Chapter 5
Logical data-set processing	Chapter 6
AWS virtual tape	Chapter 7
COPY/INCLUDE, translation units, conditional assembly	Chapter 8
Configuration and `#PRAGMA`	Chapter 9
TRACE, diagnostics, PERFORMANCE, debugging	Chapter 10
Listings and XREF	Chapter 11
Linux command-line options	Chapter 12
Exact syntax, codes, and storage maps	Appendices A–F
Complete programs	Appendix G

Chapter 2 — Source Language Fundamentals

Applies to: TSDSUTIL Version 1.0.1

This chapter defines the common source-language rules used by assembler directives, machine instructions, macros, and conditional/source-management statements.

2.1 Source Statement Format

A normal source statement contains an optional label, an operation, operands, and optional comment text. Labels begin in column 1; statements without labels are normally indented.

```
COUNT    DC    F'0'
         LA    R4,BUFFER
LOOP     CLI   0(R4),C' '
```

A line whose first character is `*` is a comment. Blank lines are permitted.

TSDSUTIL uses assembler-like source conventions but does not require traditional fixed-column card formatting.

2.2 Labels and Symbol Names

Labels define symbols for storage, executable instructions, `EQU` values, DSECT fields, and other statements that establish a location or value.

A symbol name:

- is case-insensitive;
- may contain up to 63 characters;
- begins with `A`–`Z`, `@`, `$`, `#`, or `_`;
- may contain digits after the first character.

Examples:

```
RECORD   DS    CL80
LENGTH   EQU   80
LOOP     CLI   0(R4),X'00'
```

A definition is not itself a reference to the symbol being defined. Thus `NEXT EQU BASE+4` defines `NEXT` and references `BASE`.

Undefined symbols are diagnosed during assembly. When a deferred expression cannot be resolved, TSDSUTIL reports the unresolved semantic dependency rather than merely the intermediate expression where possible.

2.3 Operation and Operand Fields

The operation identifies an assembler directive, machine instruction, macro, or source-control statement. Operation names and language keywords are case-insensitive.

The operation determines the accepted operand syntax. Examples include:

```
         LR    R3,R4
         MVC   TARGET(10),SOURCE
         $PRINT MESSAGE,L'MESSAGE
         #PRAGMA TRACE
```

Exact operand forms are documented with the applicable instruction, macro, directive, or pragma.

2.4 Comments

A line beginning with `*` in column 1 is a comment:

```
* This is a comment
```

Comments do not define symbols or generate executable operations.

2.5 Statement Continuation

A trailing `+` continues the logical statement on the next physical source line. The continuation convention applies to ordinary assembler statements and `#PRAGMA` statements.

```
OPCODE VALUE="abcd"+
       "efgh"
```

The physical line break and leading white space on the continuation record are removed. Adjacent quoted fragments using the same delimiter are concatenated, so the example above supplies `VALUE="abcdefgh"`.

Continuation does **not** insert punctuation. For example:

```
OPCODE VALUE="abcd"+
       FUNC=4
```

forms the invalid logical operand `VALUE="abcd"FUNC=4`. A required comma must therefore appear explicitly before the continuation `+`.

Malformed logical statements are rejected during assembly. Date/time macro operand syntax uses TSDS6040E for this class of malformed macro statement rather than the generic invalid-instruction-operand diagnostic.

2.6 Case Sensitivity

Language keywords and symbols are case-insensitive:

```
RECORD
Record
record
```

all identify the same symbol. This manual generally uses uppercase assembler operations, macros, registers, and symbols for readability.

Quoted data is not case-folded. `C'Mickey'` retains the characters supplied by the programmer. Host values follow host rules; Linux path names, for example, remain case-sensitive.

2.7 Expressions

The common assembler-expression grammar is used by `EQU`, addresses, constants, lengths, macro operands, and other assembly-time values. Conditional-assembly expressions use the separate Boolean/comparison grammar described in Chapter 8.

Term or operator	Meaning	Example
decimal integer	Non-relocatable decimal value	`100`
`X'...'`	Hexadecimal self-defining value	`X'FF'`
`B'...'`	Binary self-defining value	`B'1010'`
`C'...'`	1–4 characters translated through EBCDIC CP037 and treated as a self-defining integer	`C'AB'`
symbol	Value of a defined or deferred symbol	`BUFFER`
`*`	Current assembly location	`*-STARTPT`
`L'symbol`	Length attribute of a symbol; no space is permitted after `L'`	`L'RECORD`
`(expr)`	Parenthesized expression	`(COUNT+1)*4`
unary `+` / `-`	Sign	`-DELTA`
`*` / `/`	Multiply or divide	`COUNT*4`
`+` / `-`	Add or subtract	`BUFFER+4`

Precedence is conventional: unary operators, then multiplication/division, then addition/subtraction. Parentheses override precedence. Arithmetic is evaluated as signed 64-bit values during expression processing, subject to the range required by the statement that consumes the result.

Relocatable arithmetic follows assembler-style rules. A relocatable value may be adjusted by a non-relocatable value. Two relocatable values cannot be added. Subtracting two relocatable values is permitted only when they belong to the same

relocation domain and produces a non-relocatable result. A relocatable value cannot be subtracted from a non-relocatable value, and multiplication/division require non-relocatable operands.

Forward references can require deferred resolution. For example:

```
FIRST    EQU   SECOND+4
         ...
SECOND   EQU   100
```

V93 and later resolve deferred `EQU` definitions to a fixed point, allowing multi-level forward chains independent of fixup-list order. If a chain depends on genuinely missing symbols, TSDS4001E recursively follows retained expression dependencies, deduplicates repeated unresolved names, and reports root missing symbols in first-reference traversal order. Dependency cycles are guarded against unbounded recursion and remain unresolved.

2.8 Literals

A literal references a constant without a separately named storage definition:

```
         CLC   FIELD,=C'ABC'
         CLC   FIELD,=X'010203'
```

TSDSUTIL allocates literal storage as required. Literal contents are data; symbol-like text inside a literal is not treated as a symbol reference. Appendix C summarizes supported constant and literal types.

2.9 DATA, START, and END

TSDSUTIL separates data definition from executable code.

```
COUNT    DC    F'0'
BUFFER   DS    CL80
*
MAIN     START ,
         ...
         $EOJ  0
         END   ,
```

The rules are:

Statement	Purpose
Data definitions before `START`	Allocate/describe program data
`START`	Begins executable program code and establishes an entry point
`END`	Marks the end of the source program
`$EOJ`	Executable macro that normally terminates execution and establishes a completion code

Ordinary data definitions cannot resume after `START`. Literals referenced by executable instructions are allocated by the assembler as required. `DS 0type` has a limited code-label use described in Chapter 3.

Program DATA and CODE occupy separate regions of TSDSUTIL virtual storage; Chapter 3 explains the programmer-facing addressing model and Appendix F gives the exact map.

2.10 Source Inclusion and Translation Units

TSDSUTIL provides two source-inclusion mechanisms with intentionally different namespace behavior:

Facility	Translation unit	Typical use
`#COPY`	Current translation unit	Common declarations, DSECTs, constants, shared source fragments
`#INCLUDE`	New translation unit	Separately assembled reusable modules

Search paths can be supplied with the command-line `-Lpath` option. Listing and XREF information retain main listing line numbers for incorporated source.

Chapter 8 is authoritative for `#COPY`, `#INCLUDE`, translation-unit namespaces, ENTRY/VCON linkage, and conditional assembly.

Chapter 3 — Defining and Addressing Data

Applies to: TSDSUTIL Version 1.0.1

TSDSUTIL uses assembler-style declarations to define constants, reserve storage, describe record layouts, and assign symbolic values. Ordinary data definitions precede the first `START`; literals are allocated by the assembler as required.

3.1 DC and DS

`DC` allocates initialized storage; `DS` reserves storage without defining an application value.

```
NAME     DC    C'MICKEY'
FLAG     DC    X'01'
COUNT    DC    F'100'
AMOUNT   DC    PL5'202608'
BUFFER   DS    CL80
WORK     DS    XL16
```

The label identifies the first byte of the item. More than one definition can be used to construct a record.

`DS 0type` performs the type's alignment and defines a location without reserving bytes:

```
TABLE    DS    0F
```

After `START`, `DS 0type` may also establish an aligned code label without adding data storage:

```
EOF      DS    0H
         $EOJ  0
```

This exception does not permit ordinary data definitions after `START`.

3.2 Data Types, Lengths, and Alignment

The principal types are summarized below; Appendix C is the detailed DC/DS reference.

Type	Meaning	Length / alignment notes
`C`, `CLn`	EBCDIC character data	Byte-oriented; `CLn` is exactly `n` bytes
`X`, `XLn`	Hexadecimal byte data	Byte-oriented; `XLn` is exactly `n` bytes
`B`	Binary bit-string data	Byte-oriented
`H`	Signed halfword integer	Natural 2-byte alignment
`F`	Signed fullword integer	Natural 4-byte alignment
`D`, `FD`	Doubleword integer forms	Natural 8-byte alignment
`A`	Address constant	Address/fullword alignment
`AL1`–`AL4`	Explicit-length address value	Exactly 1–4 bytes; no natural alignment requirement
`P`, `PLn`	Packed decimal	Byte-oriented; `PLn` is exactly `n` bytes

A repeat factor allocates the complete item repeatedly:

```
ARRAY    DS    10F
```

A zero repeat allocates no bytes. `$DFMT` displays a repeated DSECT field once and omits zero-repeat fields.

Natural alignment can insert padding before H/F/D/FD/A items. For example, a fullword following a single byte can be advanced to a fullword boundary. `DS 0type` can be used when only the alignment/location is required.

3.3 Character, Hexadecimal, Binary, and Packed Data

Character constants are EBCDIC using TSDSUTIL's default CP037 encoding:

```
TEXT     DC    C'HELLO'
NAME     DC    CL10'MICKEY'
```

An explicit character length is the field length; a shorter initializer is padded appropriately.

Hexadecimal and binary forms are useful when the representation matters:

```
FLAGS    DC    X'80'
SAVEAREA DS    XL72
BITS     DC    B'10110000'
```

Packed decimal uses `P` or `PLn`:

```
VALUE1   DC    P'123'
VALUE2   DC    PL4'-20'
```

`P` chooses the required packed length; `PLn` occupies exactly `n` bytes. A short value is padded on the high-order side; excess high-order digits are truncated when an explicit packed length is too small. The low-order nibble contains the sign.

Examples:

```
P'1'       -> X'1C'
PL2'1'     -> X'001C'
PL1'123'   -> X'3C'
```

Program modification can create invalid packed data. Diagnostic services such as `$DFMT` report the invalid representation rather than assuming it is valid decimal.

3.4 Address Constants

`A` stores the address represented by an expression:

```
PTR      DC    A(BUFFER)
```

Explicit-width forms store exactly the requested number of bytes:

```
SHORT    DC    AL2(BUFFER)
ADDR3    DC    AL3(BUFFER)
ADDR4    DC    AL4(BUFFER)
```

TSDSUTIL program addressing is 24-bit while general registers are 32 bits. An address value and an ordinary 32-bit register integer should therefore not be confused.

3.5 EQU and ORG

`EQU` assigns a symbolic value without allocating storage; an `EQU` symbol has length zero.

```
BUFLEN   EQU   80
GPR1     EQU   1
NEXT     EQU   BASE+4
FRED     EQU   4
```

Where a register operand is expected, `FRED` above is equivalent to register value 4 (`R4`). An `EQU` can use a forward reference and be resolved during later assembly processing. Inside a DSECT, an `EQU` may be relocatable where appropriate but occupies no layout bytes.

`ORG` changes the current data-definition location. It is available only while defining data or a DSECT and is useful for overlays or alternate interpretations of the same storage. Once `START` begins executable code, `ORG` cannot resume or rearrange ordinary data definitions.

3.6 General Registers

TSDSUTIL provides sixteen 32-bit System/370 general registers, with predefined symbols `R0` through `R15`.

A programmer can define any suitable register alias with `EQU`:

```
RECPTR   EQU   4
COUNTREG EQU   5
*
         LA    RECPTR,RECORD
         L     COUNTREG,COUNT
```

R0 has the normal System/370 special meaning in an addressing field where encoded register zero means that no base or index participates. When R0 is an actual register operand, it is general register zero.

3.7 Storage Operands and Effective Addresses

A storage operand can use a symbolic location or the displacement/register form accepted by the instruction or service.

```
COUNT    DC    F'0'
RECORD   DS    CL80
*
         L     R3,COUNT
         LA    R4,RECORD
         CLI   5(R4),C' '
         MVC   0(10,R4),SOURCE
```

Common System/370 forms are:

Form	Components
`D1(X1,B1)`	displacement, optional index, optional base
`D2(B2)`	displacement and optional base

A label names a symbolic location known at assembly time; a displacement/register operand computes its effective address from runtime register contents. Individual macro entries state which address forms they accept.

TSDSUTIL does not require a base register for executable CODE. It does not assemble the script into physical System/370 machine code and therefore does not need a traditional `USING MAIN,R12` for instruction addressability.

3.8 DSECT and USING

A DSECT describes a layout without allocating an instance of that storage:

```
RECORD      DSECT ,
FIRST_NAME  DS    CL10
LAST_NAME   DS    CL10
ADDRESS     DS    CL20
CITY        DS    CL10
STATE       DS    CL2
ZIP         DS    CL5
DATE        DS    PL5
```

Field symbols represent offsets within the DSECT. The actual bytes can come from a defined test record, GET buffer, GETMAIN area, or other valid storage.

`USING` associates a DSECT with a register containing an actual base address:

```
         LA    R4,REC1
         USING RECORD,R4
```

`DROP` removes one or more active DSECT USING associations by register number:

```
         DROP  R4
         DROP  R5,R6
```

`DROP` does not accept a label and accepts only R1 through R15. Dropping a register with no active DSECT association is harmless. TSDSUTIL restricts `USING` to DSECTs; it is not a general executable-code base-register directive.

Some diagnostic services deliberately require an explicit address rather than relying on USING. For example:

```
         $DFMT RECORD,0(R4)
```

Unnamed DSECT fields can contribute to layout but cannot be referenced by name; `$DFMT` omits them. DSECT overlays created with `ORG` are allowed, but `$DFMT` stops rather than guessing when sequential formatting becomes ambiguous.

3.9 Controlled Virtual Storage

TSDSUTIL executes in a controlled 24-bit virtual address space. The programmer-facing rules are:

Rule	Effect
LOWMEM is `000000`–`000FFF`	Read-only
Program DATA begins at `001000`	Normal program storage
CODE occupies the high `F00000` region	Executable addresses; not writable data storage
GETMAIN storage is controlled virtual storage	Subject to the same address validation
General registers are 32-bit	Program addresses remain 24-bit
Invalid/protected references	Produce TSDSUTIL runtime errors rather than host-memory access

Machine-instruction effective-address calculation follows the applicable TSDSUTIL/System/370 behavior, including 24-bit wrapping for operations such as `LA`. TSDSUTIL generally permits unaligned machine-instruction storage references unless a particular operation imposes a stronger requirement; natural DC/DS alignment is therefore primarily a layout property.

Appendix F is authoritative for the exact virtual-storage map. Chapter 4 describes instruction-level protection and addressing behavior.

Chapter 4 — System/370 Machine Instructions

Applies to: TSDSUTIL Version 1.0.1

TSDSUTIL supports a substantial subset of the IBM System/370 machine-instruction set. This chapter does not reproduce the IBM System/370 Principles of Operation. Unless documented otherwise, a supported instruction is intended to have its normal System/370 meaning. Appendix A is the authoritative supported-instruction reference.

4.1 Instruction Syntax and Formats

Machine instructions use familiar assembler syntax:

```
LR    R3,R4
L     R5,COUNT
LA    R4,RECORD
MVC   TARGET(10),SOURCE
CLI   FLAG,X'01'
BNE   NOT_SET
```

Labels can appear on executable instructions. Mnemonics and symbols are case-insensitive. The mnemonic determines the required register, immediate, storage, length, mask, and base/displacement operand forms; invalid forms are diagnosed during assembly.

TSDSUTIL implements the instruction formats required by its supported instruction set, including register-to-register and storage-oriented forms. Appendix A is authoritative for which mnemonics are implemented by the current TSDSUTIL release.

4.2 Condition Codes and Branching

TSDSUTIL maintains System/370 condition codes 0 through 3. Instructions that set the condition code do so according to their architectural operation unless specifically documented otherwise.

For `BC` and `BCR`, branch-mask bits correspond to condition codes as follows:

Condition code	Mask bit
0	8
1	4
2	2
3	1

A zero mask never branches. Extended branch mnemonics such as `BE`, `BNE`, `BL`, and `BH` use the corresponding mask semantics.

Macros can also define condition-code results; those are service behavior and are documented in the macro reference rather than here.

4.3 Addressing and Storage Protection

Machine instructions operate on TSDSUTIL virtual addresses, not host pointers. Chapter 3 introduces the programmer-facing storage model and Appendix F gives the exact map.

The important instruction-level rules are:

Rule	Instruction effect
24-bit virtual addressing	Effective program addresses are constrained to the TSDSUTIL address space
LOWMEM is read-only	Reads may be permitted; writes fail
CODE is protected from data writes	Branch/EX targets can be executable CODE; ordinary writes fail
Entire operand range must be valid	A valid starting byte is insufficient if the full operand crosses inaccessible storage
Many unaligned references are permitted	Unless the particular operation has a stronger requirement

Rule	Instruction effect
Invalid/protected reference	Runtime diagnostic and possible SYSTEM ABEND

When zero is encoded as a base or index register field, no register contributes to that part of the effective address. This is distinct from instructions that explicitly operate on general register R0.

4.4 STCK and STCKE

`STCK` stores an eight-byte IBM Time-of-Day value. On Linux, TSDSUTIL obtains host UTC time through the TSCLIB `mvs_datetime` service and converts it to IBM TOD format; a successful Linux STCK sets CC=0. On a native GNU-compatible s390/s390x build, TSDSUTIL executes hardware `STCK` and preserves the processor condition code.

Classic STCK uses the IBM TOD epoch of 1900-01-01 00:00:00 UTC. Its 64-bit epoch rolls over at 2042-09-17 23:53:47.370496 UTC. TSDSUTIL does not infer a later epoch from a classic STCK value; use STCKE when an extended epoch is required.

`STCKE` (`B278`) stores the architected 16-byte extended TOD value: a one-byte epoch index, 104 bits of TOD clock value, and a two-byte programmable field. On Linux, TSDSUTIL derives the value from host UTC; the programmable field is zero because TSDSUTIL does not implement `SCKPF`. A successful Linux STCKE sets CC=0. Native s390/s390x builds execute hardware `STCKE` and preserve its condition code.

TSDSUTIL civil-date conversions remain limited to 1900–2899. Chapter 5 and the macro reference describe the date/time conversion services.

4.5 EX

`EX` follows the System/370 rule: the low-order eight bits of its general register are ORed into bits 8–15 (the second byte) of the subject instruction, and the resulting temporary instruction is executed. The stored subject instruction is not permanently modified.

Important consequences include:

Subject	Effect of EX modification
Supported SS instruction	Can modify second-byte length information
`CLI`, `MC`, `MVI`, `NI`, `OI`, `TM`, `XI`	Can modify the immediate operand
`TS`	Bits 8–15 are ignored by TS, so the operation is unchanged
`STCK` (`B205`)	Nonzero modification can create a different `B2xx` opcode; it must itself be supported
`EX`	Cannot be the subject of another EX

For example:

```
         LA    R5,X'F0'
         EX    R5,TEST
         ...
TEST     CLI   BYTE,X'05'
```

executes the `CLI` with immediate value `X'F5'` because `X'05' OR X'F0' = X'F5'`.

The EX target must be a valid executable instruction in CODE storage, and the effective instruction after modification must remain supported by TSDSUTIL. Because CODE is protected from ordinary writes, EX is the supported architectural mechanism for temporary instruction modification.

4.6 Behaviors Worth Noting

Most supported instructions need no TSDSUTIL-specific discussion. The following behaviors are particularly useful when writing or debugging scripts:

Area	Behavior
`MVC` overlap	Moves left to right, preserving System/370 propagation behavior rather than host `memmove` semantics
R0 in address fields	Encoded zero means no base/index; explicit R0 operands still use general register zero
`TR` / `TRT`	Require the architectural 256-byte translation table
`MVCL` / `CLCL`	Addresses and lengths come from registers at execution time
`ED` / `EDMK`	Follow packed-decimal edit rules; TRACE determines the relevant source span from the pre-execution edit pattern
Unaligned storage	Generally permitted by TSDSUTIL unless the operation has an explicit stronger requirement

4.7 Runtime Errors and Debugging

Invalid instruction addresses, protected writes, inaccessible storage ranges, and other runtime storage errors are reported using TSDSUTIL diagnostics and can cause a SYSTEM ABEND. The diagnostic reports TSDSUTIL virtual addresses rather than exposing host pointers.

TRACE can record instruction execution, relevant registers, condition codes, and effective storage spans. For dynamically sized operations it records the effective participating range rather than relying only on the printed source operand. Chapter 10 is authoritative for TRACE, execution limits, diagnostics, PERFORMANCE, and debugging.

4.8 Supported Instruction Reference

Appendix A contains the complete current supported-instruction table. If an instruction is listed there and no TSDSUTIL-specific exception is documented, use its normal System/370 definition.

When evaluating compatibility with existing assembler code, check both the instruction inventory and TSDSUTIL's controlled execution environment: 24-bit virtual storage, LOWMEM/CODE protection, unaligned-reference policy, and any operating-system services on which the original program depended.

Chapter 5 introduces the TSDSUTIL macros and services that provide utility-oriented operations outside the System/370 machine-instruction architecture.

Chapter 5 — Macros and Program Services

Applies to: TSDSUTIL Version 1.0.1

TSDSUTIL macros provide utility services that are awkward or inappropriate to express as System/370 machine instructions. This chapter explains the common programming model and service families. **Appendix B is the authoritative reference for exact syntax, operands, register effects, condition codes, and service-specific return codes.**

5.1 Common Macro Conventions

A macro may have a label and may accept positional operands, keyword operands, or lists. Register names are ordinary symbols: `R4`, `4`, or an `EQU` symbol whose value is 4 can identify register 4 where a register operand is expected. Parenthesized forms are significant for macros that distinguish a register value from storage addressed through a register.

Macro services may return status in R15, other documented registers, or CC. Do not assume a common register convention beyond what Appendix B specifies.

5.2 Service Families

Family	Services	Purpose
Display/inspection	`$PRINT`, `$DUMP`, `$REGS`, `$DFMT`, `$LINENO`	Display program state and source context
Linkage/registers	`$SAVE`, `$RESTORE`, `$RETURN`	Save-stack and return services
Random/delay	`$RAND`, `$SRAND`, `$SLEEP`	Random values and host delay
Time	`$TIMEUSD`, `TIME DEC`	Current time representations
Validation/conversion	`$VALDATE`, `$VALTIME`, `$CVTDATE`, `$FMTDATE`	Validate, convert, and format date/time values
Date/time arithmetic	`$DAYDIFF`, `$DAYADJ`, `$TIMEADJ`, `$EXPDATE`	Date/time calculations
Testing	`$ASSERT`	Executable regression assertions
Dynamic storage	`GETMAIN`, `FREEMAIN`	Allocate and release virtual storage
Terminal	`WTO`, `WTOR`	Communicate through the controlling terminal
Program control	`ABEND`, `$EOJ`	Abnormal or normal termination
Data-set	`OPEN`, `CLOSE`, `GET`, `PUT`	Logical data-set processing
DCB inspection	`$SHOWDCB`, `$TESTDCB`, `$MODDCB`	Query, test, or modify DCB state
Physical tape	`$READ`, `$WRITE`, `$CONTROL`	BLP physical block I/O and positioning
Trace control	`$TRACE`	Program-visible TRACE control

5.3 Display and Test Services

`$PRINT`, `$DUMP`, `$REGS`, and `$DFMT` are intended for reports, diagnostics, and debugging. `$DFMT` interprets an address using DSECT metadata. `$ASSERT` compares a register or storage value with a literal and causes a SYSTEM EEE ABEND when the assertion fails; success has no program-visible side effects. Chapter 10 describes their use in a debugging workflow.

On Linux, `$PRINT`, WTO, and WTOR convert EBCDIC text to ASCII for display. Converted bytes outside the printable ASCII range are emitted as `.`; the original message length is preserved.

5.4 Save/Restore and Return

`$SAVE` and `$RESTORE` use the TSDSUTIL save stack. `$RETURN` restores the requested register list and returns through the saved linkage state. These are emulator services rather than generated machine-code macro expansions. Exact list syntax and failure behavior are in Appendix B and Appendix D.

5.5 Dynamic Storage

`GETMAIN` allocates from the GETMAIN region and `FREEMAIN` releases an exact allocation. R forms treat failure as fatal; RC forms permit the program to test the returned status. Debug fill and allocation diagnostics are configuration-controlled and are described in Chapters 9 and 10.

5.6 Data-Set and Tape Services

`OPEN`, `CLOSE`, `GET`, and `PUT` operate on DCBs and logical records. Chapter 6 is authoritative for their programming model. AWS tape adds label and volume semantics described in Chapter 7.

`$READ`, `$WRITE`, and `$CONTROL` are physical BLP tape services. `$CONTROL` supports `BSB`, `BSF`, `FSB`, `FSF`, and `REWIND`; the spacing operations accept `COUNT=value` or `COUNT=(register)`. These services share the physical-tape return-code model in Appendix D; Chapter 7 explains tape marks, BOT, EOI/EOT, and positioning semantics.

5.7 Date and Time Services

The date/time family uses common representations and return-code conventions. The principal representations are JDATE, GDATE, STIME, STCK, and STCKE. Civil-time conversions can use configured timezone/DST rules; timestamp-to-timestamp conversion remains UTC. Appendix B gives exact macro syntax and Appendix D gives the shared return-code table.

5.8 Program Termination and Terminal Services

`$EOJ` performs normal program termination with a completion code. `ABEND` performs abnormal termination. WTO/WTOR use the controlling Linux terminal rather than ordinary redirected stdin/stdout. Appendix D is authoritative for completion and ABEND behavior.

For quick recognition, the five physical-positioning forms are `TYPE=BSB`, `TYPE=BSF`, `TYPE=FSB`, `TYPE=FSF`, and `TYPE=REWIND`. The shared physical-tape namespace uses R15=`40` for a BOT boundary reached during backward positioning.

Chapter 6 — Data Set Processing

Applies to: TSDSUTIL Version 1.0.1

TSDSUTIL presents Linux files and supported tape data sets through an IBM-style DCB and logical-record interface. This chapter owns the logical DCB/OPEN/GET/PUT programming model; Chapter 7 adds AWS-specific tape semantics and Chapter 9 owns complete `#PRAGMA DD` syntax.

6.1 DDNAME and DCB

A DD definition associates a DDNAME with an external resource. A DCB is the program-visible control block used by OPEN, CLOSE, GET, and PUT.

DCB attribute	Purpose
`DDNAME`	Selects the DD allocation
`MACRF`	Permitted logical/physical access
`RECFM`	Record organization
`LRECL`	Logical record length
`BLKSIZE`	Physical block size
`EODAD`	Optional logical end-of-data branch target
`OPENEXIT`	Optional executable OPEN-exit address; zero disables the exit

A DCB can contain attributes assembled in the source and can acquire effective attributes from the DD allocation during OPEN. CLOSE restores the baseline DCB state.

6.2 Logical Record Formats

RECFM family	Logical model
`F`, `FA`, `FM`	One fixed logical record per block
`FB`, `FBA`, `FBM`	Multiple fixed logical records per block
`V`, `VA`, `VM`	Variable logical records
`VB`, `VBA`, `VBM`	Blocked variable logical records
`U`	Undefined physical block treated as the record unit where supported

The A/M suffix is retained as part of the effective RECFM. Exact validation rules are part of the Phase 4 code audit and the reference material.

6.3 DEVTYPE

`DEVTYPE ddname,return-area` queries the allocation resource class without opening a DCB. Both operands are required and positional. `ddname` addresses an 8-byte blank-padded DDNAME and `return-area` addresses an 8-byte writable area; normal storage-address forms and structural `(Rx)` runtime-address forms are accepted.

The complete return area is cleared before allocation lookup. R15=0 means the DD is defined; R15=4 means it is undefined. Invalid operand storage is an addressing failure, not R15=4.

Allocation type	8-byte result
3420-style AWS tape	`X'3210800300007FF8'`
DATA / simulated SYSIN	`X'0000010200007FF8'`
Linux PATH	`X'0000010300007FF8'`
defined DUMMY	`X'0000000000000000'`

A zero DEVTYPE value for DUMMY is therefore distinguished from an undefined DD by R15.

6.4 RDJFCB

`RDJFCB dcb,jfcb-area` returns a zero-initialized 176-byte JFCB-compatible image for the DD currently named by a valid DCB. Both operands are required and positional and accept the established address forms, including structural `(Rx)` runtime addresses. R15=0 means the image was returned; R15=4 means the DCB is valid but its DDNAME is blank, unusable, or not currently allocated. Invalid DCB/result-area storage is an addressing failure.

Only the supported JFCB fields are populated: `JFCBDSNM`, `JFCBLTYP`, `JFCBFLSQ`, `JFCDSORG`, `JFCRECFM`, `JFCBLKSI`, `JFCLRECL`, `JFCBNVOL`, and `JFCBVOLS`; all other bytes remain zero. `#COPY $_JFCB` supplies the built-in 176-byte DSECT and `JFCBLEN=176`.

PATH uses the host path as `JFCBDSNM`; a value longer than 44 characters is represented as `...` plus its final 41 characters. DATA uses its stable definition-order name `SYSIN.SI001` through `SYSIN.SI999`. DUMMY uses `NULLFILE`. Tape supplies its applicable data-set name, label type, file sequence, volume serial, DSORG, and record attributes.

RDJFCB is allocation-oriented before OPEN. A successful OPEN then synchronizes the persistent JFCB record attributes (and tape sequence where applicable) to the final effective DCB. CLOSE restores the normal DCB baseline but does not roll this persistent JFCB state back.

6.5 OPEN, Attribute Merge, and OPENEXIT

OPEN associates the DCB with its DD resource, validates the requested access, and establishes effective DCB attributes. A DCB may specify `OPENEXIT=<address>`; zero or omission means that no OPEN exit is installed.

When an OPEN exit is present, TSDSUTIL first merges attributes that were actually supplied by the applicable sources, but it does **not** yet apply the normal generic OPEN defaults. Existing labeled tape contributes label-derived attributes before the exit. PATH, DUMMY, and new/unlabeled output tape contribute DD attributes plus program-DCB attributes, so an omitted RECFM, LRECL, or BLKSIZE remains zero at exit entry. DATA is intentionally different: its allocation defines FB/80/3280 when those fields are omitted, so those DATA allocation values exist before the exit and then participate in the normal program-DCB precedence.

The exit receives `R1=DCB address` and `R14=X'00FFFFFE'`. It may use `$MODDCB` to change only RECFM, LRECL, and BLKSIZE and returns normally with `BR R14`; R15 has no OPENEXIT return-code meaning. After return, TSDSUTIL supplies its generic defaults only to fields still zero, performs the normal final DCB validation, updates the persistent JFCB, and completes OPEN. Consequently RDJFCB after OPEN, and after the later CLOSE, reports the final OPENEXIT-adjusted attributes.

`X'FFFFFE'` is a reserved OPENEXIT return sentinel. Reaching it outside an active OPENEXIT causes SYSTEM EEE. While an exit is active, OPEN, CLOSE, RDJFCB, GET, PUT, `$READ`, `$WRITE`, `$CONTROL`, and `$EOJ` are prohibited, as are `$MODDCB` changes to DDNAME or OPENEXIT; these protected-environment violations cause SYSTEM EEE. DEVTYPE, GETMAIN/FREEMAIN, WTO/WTOR, `$PRINT`, DUMP, permitted `$MODDCB` changes, and deliberate ABEND processing remain available.

DDNAME-not-allocated (`TSDS8021`) is the recoverable OPEN case returned through R15. Other OPEN failures are fatal SYSTEM EEE ABENDs under the current OPEN policy. Appendix D is authoritative for the policy.

6.6 GET and End of Data

GET reads one logical record into program storage. With `EODAD`, logical EOF transfers control to that address. Without `EODAD`, EOF is returned in R15 so the program can handle it directly.

For blocked formats, GET performs unblocking. The application continues to see one logical record per GET.

6.7 PUT

PUT writes one logical record from program storage. For blocked formats, TSDSUTIL performs the required blocking and flushes pending data as required by CLOSE. The application supplies logical records rather than physical blocks.

For Linux host-file output using a fixed RECFM family, a DD may specify `TRIM=YES`. The record remains a full-LRECL fixed record inside TSDSUTIL, but trailing EBCDIC blanks are omitted when the host text line is written. `TRIM=NO` is the default. The option does not change fixed-format INPUT padding and has no effect on tape or variable-format output.

6.8 CLOSE and Cleanup

CLOSE completes outstanding output processing, releases the DCB/resource association, and restores baseline DCB attributes. It accepts either a bare DCB or a parenthesized list; `(REWIND)` and `(FREE)` may follow a DCB in a list and are accepted case-insensitively. TSDSUTIL also closes open DCBs during normal end-of-job processing and performs cleanup during ABEND as described in Appendix D.

6.9 DCB Inspection and Modification

Service	Use
`$SHOWDCB`	Return selected effective DCB fields to program storage
`$TESTDCB`	Compare DCB state/attributes and report comparison status
`$MODDCB`	Change supported baseline fields while closed; during OPENEXIT, change only effective RECFM/LRECL/BLKSIZE

Appendix B gives exact field lists, syntax, and register/CC effects.

6.10 Host-File Pattern

```
#PRAGMA DD SYSIN,PATH='./input.dat',RECFM=FB,LRECL=80,BLKSIZE=800
SYSIN    DCB   DSORG=PS,MACRF=GM,DDNAME=SYSIN,EODAD=EOF
RECORD   DS    CL80
         START ,
         OPEN  (SYSIN,INPUT)
LOOP     GET   SYSIN,RECORD
*        Process RECORD.
         B     LOOP
EOF      CLOSE (SYSIN)
         $EOJ  0
         END   ,
```

Appendix G contains complete read/write examples. The same logical model applies to supported AWS tape data sets after tape-specific OPEN processing.

Chapter 7 — AWS Virtual Tape Processing

Applies to: TSDSUTIL Version 1.0.1

TSDSUTIL uses TSCLIB/VTAPE to process AWS virtual tape images. Tape DCBs use the same OPEN/CLOSE and logical GET/PUT model as Chapter 6, with additional volume, label, data-set, expiration, and physical-positioning rules.

7.1 Tape DD Model

A tape DD identifies an AWS image and supplies tape-specific selection information. Chapter 9 is authoritative for exact DD syntax.

Attribute	Meaning
`TAPE`/path	AWS image
`VOL`	Expected volume serial
`DSN`	Tape data-set name
`LABEL`	`SL`, `NL`, or `BLP`
`FILE`/label number	Tape-file selection where applicable
`EXPDT` / `RETPD`	Output retention/expiration control
DCB attributes	RECFM, LRECL, BLKSIZE, MACRF

Volumes are mounted lazily and can be shared by compatible DCBs. The implementation maintains one physical tape position per mounted volume.

7.2 Label Modes

Mode	Interpretation	Typical access
`SL`	Standard IBM labels are interpreted/created	Logical GET/PUT
`NL`	No standard labels	Restricted; see reference rules
`BLP`	Bypass label processing	Physical `$READ/$WRITE/$CONTROL`

SL processing validates volume/data-set identity and obtains or verifies effective DCB information from labels. BLP intentionally exposes physical blocks and tape marks instead of logical data sets.

7.3 Logical Tape Processing

After successful OPEN, GET and PUT use the Chapter 6 logical-record model. Fixed, blocked-fixed, variable, blocked-variable, and undefined formats are handled according to the effective DCB. The application normally does not process AWS block headers or tape marks itself.

INPUT positioning begins at the selected data set. OUTPUT positioning and label creation follow the selected tape mode and output policy. CLOSE completes label/output processing and releases the volume when the final DCB using it closes.

7.4 Expiration Protection

Output processing honors supported expiration/retention information. `EXPCHK=BYPASS` is an explicit override and should be used deliberately. Exact accepted EXPDT/RETPD forms and failure behavior are in the reference material.

7.5 Logical versus Physical Services

Interface	Unit seen by program	Label interpretation	Typical MACRF
GET/PUT	Logical record	Yes for labeled processing	GM/PM
`$READ/$WRITE`	Physical tape block or mark	Bypassed	R/W/RW
`$CONTROL`	Physical tape position	Bypassed	R/W/RW

Do not mix the logical-record model with assumptions about physical AWS blocks.

7.6 Physical $READ and $WRITE

$READ returns one physical data block or a physical boundary status. On a successful data read, R1 contains the full physical block length. $WRITE writes a physical data block; TYPE=TM writes a tape mark.

If a physical operation is attempted with an incompatible DCB access mode, TSDSUTIL retains RC=24 in R15 and emits a runtime diagnostic that names both the actual MACRF and the required physical mode. For example, $READ against MACRF=GM reports invalid DCB access mode: MACRF=GM; requires R or RW.

The exact shared physical-tape return-code table is in Appendix D.

7.7 $CONTROL Positioning

Physical BLP positioning supports:

TYPE	Operation
BSB	Backspace physical block(s)
BSF	Backspace tape-file boundary/boundaries
FSB	Forward-space physical block(s)
FSF	Forward-space tape-file boundary/boundaries
REWIND	Rewind to physical BOT

COUNT= is optional for BSB, BSF, FSB, and FSF, defaults to one, and may be an ordinary expression or structural runtime-register value. It is not valid with REWIND. Repeated positioning stops at the first nonzero status.

$CONTROL changes R15 and preserves CC. It operates on the VTAPE/AWS position only; it does not interpret RECFM, logical records, labels, or data-set contents.

For BSB/FSB, crossing a tape mark before a data block returns the shared tape-mark status. BSF/FSF return success when the requested tape-file boundary is found. Forward positioning can reach physical end-of-image; backward positioning can reach BOT. REWIND and successful backward repositioning clear remembered EOT state so processing can resume after an earlier end condition.

7.8 Physical Tape Status Model

The shared namespace distinguishes successful completion, logical/physical boundary conditions, invalid state, host/tape errors, format errors, resource/internal errors, and BOT. Appendix D owns the exact numeric values. Programs using physical services should test the documented status rather than treating every nonzero R15 as the same failure.

7.9 TRACE and Diagnostics

TRACE records OPEN/CLOSE/GET/PUT/READ/WRITE/CONTROL activity with relevant DD/tape context. $CONTROL detail includes DDNAME, DSN, VOLSER, path, control type, boundary condition, and R15 when available. Chapter 10 explains TRACE; Appendix E owns diagnostic codes.

7.10 Example

Appendix G contains both a standard-labeled logical tape copy and a BLP physical example using $READ and $CONTROL.

Physical Tape Control status examples

In the shared namespace, RC=16 identifies physical end-of-image reached during forward positioning and RC=40 identifies BOT reached before a backward positioning request can complete. See Appendix D for the complete table.

Chapter 8 — Conditional Assembly and Translation Units

Applies to: TSDSUTIL Version 1.0.1

This chapter is authoritative for conditional definitions, `#COPY`, `#INCLUDE`, translation-unit namespaces, and ENTRY/VCON linkage.

8.1 Conditional Definitions

Definitions can originate from several layers. Higher-precedence definitions replace lower-precedence values:

`BUILTIN < SYSTEM < PRAGMA < CLI < SOURCE`

Source `#DEFINE` therefore has the highest precedence. `#UNDEFINE name` removes the currently active definition record regardless of the origin that supplied it. TSDSUTIL keeps one active value per name; removing an overridden name does **not** reveal an older lower-precedence BUILTIN, SYSTEM, PRAGMA, or CLI value. Chapter 9 describes configuration sources and Chapter 12 describes CLI syntax; the semantic precedence rule lives here.

8.2 Conditional Assembly

Directive	Purpose
`#IF expr`	Begin conditional region
`#ELIF expr`	Test another branch
`#ELSE`	Select fallback branch
`#ENDIF`	End conditional region
`#DEFINE name[=value]`	Define at SOURCE precedence
`#UNDEFINE name`	Remove the currently active definition entirely

A bare name tests definedness. Conditional expressions support `!`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `&&`, `||`, and parentheses. Conditional regions may be nested subject to implementation limits.

Conditional assembly selects source before ordinary assembly; it is not runtime branching.

8.3 `#COPY` versus `#INCLUDE`

Property	`#COPY`	`#INCLUDE`
Translation unit	Current TU	New TU
Symbol namespace	Shared with caller	Separate
Typical use	Common definitions/source fragments	Separately linked module
ENTRY/VCON boundary	No new boundary	Yes

Use COPY when the inserted text should behave exactly as though it appeared in the current source. Use INCLUDE when the source should have its own namespace and participate in final TU linkage.

8.4 Translation-Unit Namespaces

Labels are local to their translation unit except for explicitly published linkage. Duplicate local names can therefore exist in separate included TUs. Listings/XREF use TU identification where needed to disambiguate them.

Ordinary symbol references resolve within the applicable TU namespace. Cross-TU references use the ENTRY/VCON mechanism rather than implicitly searching every namespace.

8.5 ENTRY and VCON

`ENTRY` is a labeled, operand-free directive. The label on `ENTRY` is the public linkage name. It publishes the relocatable DATA or CODE location defined by the following statement or, when the next statement is unlabeled, the current DATA/CODE location. `ENTRY` is not valid inside a DSECT.

```
PUBLIC   ENTRY ,
TARGET   DS    F
```

The example publishes `TARGET` under the global linkage name `PUBLIC`. For a CODE entry point, place the labeled `ENTRY` immediately before the first instruction or other CODE location to be exported.

A `V(symbol)` constant creates a VCON resolved during final linkage against a published ENTRY. VCONs can therefore contain linked addresses of exported DATA or CODE symbols in another translation unit. Final linkage validates unresolved VCON/ENTRY relationships after the participating translation units have been assembled. V93 deferred-symbol reporting preserves the originating source context for unresolved expressions.

8.6 Search and Inclusion Rules

COPY and INCLUDE use the configured source-search mechanism. Recursive active inclusion and excessive nesting are diagnosed. The default COPY/INCLUDE nesting limit is 32 and is configurable from 1 through 1024.

`#INCLUDE` is valid only while that translation unit's include window is open—that is, before DATA or CODE statements have begun in the TU. A newly included TU begins with its pragma preamble disabled, so `#PRAGMA` statements are not permitted inside an included translation unit. COPY remains part of the current TU and follows that TU's existing state.

Conditional directives can surround COPY/INCLUDE statements, and copied/included source can itself contain conditional assembly subject to normal nesting and definition rules.

8.7 Choosing the Facility

Need	Use
Reuse common constants, DSECTs, or source fragments in one namespace	`#COPY`
Build a separately named/linkable source unit	`#INCLUDE`
Publish a relocatable DATA or CODE location	labeled `ENTRY`
Store a linked DATA or CODE address from another TU	`V(symbol)`
Select source by build/configuration definition	`#IF` family

Appendix G includes a compact separate-TU ENTRY/VCON example. Chapter 11 explains how translation units appear in listings and XREF.

Chapter 9 — Configuration and #PRAGMA

Applies to: TSDSUTIL Version 1.0.1

TSDSUTIL uses `#PRAGMA` statements for assembly, execution, listing, debugging, storage, date/time, and DD-allocation controls. The same pragma language is accepted in the source preamble and Linux configuration files.

This chapter defines the configuration interface. Chapters 6–8 and 10–11 explain the facilities controlled by these settings.

9.1 Source Preamble and Continuation

Source pragmas must precede the first ordinary source statement. Blank lines and comments may occur in the preamble; a later `#PRAGMA` is an assembly error.

```
#PRAGMA LISTING_WIDTH 160
#PRAGMA XREF
#PRAGMA DD SYSIN,PATH='./input.dat',RECFM=FB,+
 LRECL=80,BLKSIZE=800
*
INPUT    DCB   DSORG=PS,MACRF=GM,DDNAME=SYSIN,EODAD=EOF
```

A trailing `+` continues a pragma. The continuation line does not repeat `#PRAGMA`.

9.2 General Pragmas

Boolean options use `#PRAGMA OPTION` and `#PRAGMA NO_OPTION`. Valued options use `#PRAGMA OPTION value`; they do not use `NO_`.

Pragma	Default / range	Purpose	Details
`CODE_LISTING`	Off	Include generated code/instruction information in the listing	Ch. 11
`RUNTIME_LISTING`	On	Copy runtime program output into the listing	Ch. 11
`RUNTIME_RC_MESSAGES`	On	Emit warning lines for supported recoverable runtime service RCs	Ch. 10
`PERFORMANCE`	Off	Produce the detailed end-of-run performance profile	Ch. 10
`EXECUTE`	On	Execute after successful assembly	Ch. 10
`TRACE`	Off	Enable normal TRACE processing	Ch. 10
`TRACE_ACTIVE`	Off	Select active TRACE mode; disable with `NO_TRACE`	Ch. 10
`GETMAIN_DEBUG`	Off	Report GETMAIN/FREEMAIN allocation activity	Ch. 10
`XREF`	Off	Produce cross-reference output	Ch. 11
`XREF_LONG`	Off	Include normally omitted XREF information such as unreferenced symbols	Ch. 11
`LISTING_WIDTH n`	132; 132–255	Listing/report width	Ch. 11
`MAX_INSTRUCTIONS n`	10,000,000; `0` = unlimited	Runtime execution limit	Ch. 10
`MAX_SOURCE_DEPTH n`	32; 1–1024	COPY/INCLUDE nesting limit	Ch. 8
`MAX_CONDITIONAL_DEPTH n`	64; 1–1024	Conditional-assembly nesting limit	Ch. 8

Pragma	Default / range	Purpose	Details
`ABEND_STATUS n`	100; 0–255	Linux process status used for ABEND	App. D
`PARM 'text'`	none; max 100 bytes	Startup parameter string	Ch. 5
`LISTING_FILE='path'`	none on Linux	Select listing host file; configuration files only	Ch. 11
`TRACE_FILE='path'`	none on Linux	Select TRACE host file; configuration files only	Ch. 10
`GETMAIN_FILL X'xx'`	`X'00'`	Initialization byte for new GETMAIN storage	Ch. 10

`MAX_INSTRUCTIONS 0` forces PERFORMANCE off even if profiling was requested elsewhere. `NO_XREF_LONG` and `NO_TRACE_ACTIVE` are not separate modes; use `NO_XREF` or `NO_TRACE` respectively. On Linux, TRACE has no implicit output file: `TRACE/TRACE_ACTIVE` enables the mode, but effective trace output requires a usable `TRACE_FILE` or `--trace-file=` destination. If that destination cannot be opened, tracing remains effectively disabled and startup continues.

9.3 Linux output filename templates

`LISTING_FILE` and `TRACE_FILE` are Linux host configuration pragmas and are valid only in `/etc/tsdsutil.conf` and `$HOME/.config/tsdsutil.conf`; they are not valid in a source-program pragma preamble. Hyphenated spellings `LISTING-FILE` and `TRACE-FILE` are accepted as aliases.

When primary input is selected by `--script`/`-s`, `%s` in the selected filename expands to the basename of the actual script file opened with its final extension removed. `%%` emits a literal percent sign. Other percent escapes are invalid. Explicit CLI `--listing=` and `--trace-file=` filenames have highest precedence and are literal; template processing is not applied to them.

```
#PRAGMA LISTING_FILE='./list/%s.lst',STDIN=SKIP
#PRAGMA TRACE_FILE='./trace/%s.trace',STDIN=SKIP
```

If stdin is the primary source and a filename candidate contains `%s`, `STDIN=SKIP` causes that particular pragma candidate to be ignored. A lower-precedence configuration value may therefore remain effective. Without `STDIN=SKIP`, an unresolved `%s` is a configuration error. `STDIN=SKIP` has no effect on a literal filename that does not contain `%s`. Output directories are never created automatically and must already exist.

9.4 Conditional Definitions

Pragma definitions use:

```
#PRAGMA DEFINE DEBUG
#PRAGMA DEFINE LEVEL=2
#PRAGMA DEFINE ENV = TEST
```

They have PRAGMA precedence. The complete conditional-definition hierarchy is:

```
BUILTIN < SYSTEM < PRAGMA < CLI < SOURCE
```

`#DEFINE` and `#UNDEFINE` are source-language directives rather than pragmas and are described with conditional assembly in Chapter 8.

9.5 Virtual-Storage Region Sizes

The movable lower-storage regions are configured with:

```
#PRAGMA DATA_SIZE 64K
#PRAGMA CONTROL_SIZE 4K
#PRAGMA LITERAL_SIZE 8K
```

Accepted values are positive whole-KiB quantities. Suffixes include `K`, `KB`, `KiB`, `M`, `MB`, and `MiB`; an unsuffixed byte value must be a multiple of 1024.

These settings change region sizes, not the fixed architectural boundaries: LOWMEM remains `000000-000FFF`, DATA begins at `001000`, CODE remains `F00000-FFFFFF`, and GETMAIN receives the remaining storage below CODE. CONTROL

follows DATA and LITERAL follows CONTROL. An invalid size or completed layout is TSDS7002E. See Appendix F for the complete virtual-storage map.

9.6 DD Allocation

`#PRAGMA DD` establishes an external allocation before assembly depends on it.

Linux host files

```
#PRAGMA DD INPUT,PATH='./input.dat',RECFM=FB,+
 LRECL=80,BLKSIZE=8000
```

Linux in-stream DATA

```
#PRAGMA DD,SYSIN,DATA[,DLM=cc][,RECFM=F|FB|V|VB][,LRECL=n][,BLKSIZE=n]
```

The records immediately following the pragma are captured as immutable logical input. Without `DLM=`, the first physical line beginning with `#` ends DATA and is then processed normally; `#PRAGMA COMMENT ...` is therefore a convenient visible terminator. With `DLM=cc`, the two-character delimiter must appear in columns 1-2 and be followed by end-of-line or a blank. DATA is input-only, uses normal Linux text conversion/record rules, and restarts at its first record after CLOSE followed by another OPEN INPUT. Defaults are `RECFM=FB,LRECL=80,BLKSIZE=3280`.

Each DATA definition receives a stable internal sequence number used by RDJFCB synthetic names `SYSIN.SI001` through `SYSIN.SI999`; a 1000th DATA definition is rejected. For DEVTYPE, DATA is a simulated SYSIN resource and returns `X'0000010200007FF8'`.

Linux DUMMY

```
#PRAGMA DD,NULLDD,DUMMY[,RECFM=F|FB|V|VB][,LRECL=n][,BLKSIZE=n]
```

DUMMY participates in normal DD/DCB merge and OPEN validation. OPEN INPUT succeeds and GET immediately follows the established logical EOF/EODAD path. OPEN OUTPUT or EXTEND succeeds for a logical-output DCB; PUT validates the record normally (including variable-record RDW validation) and then discards it. DUMMY does not support physical `MACRF=R`, `W`, or `RW`, nor `$READ`, `$WRITE`, or `$CONTROL`. DEVTYPE returns eight binary zero bytes with success status for a defined DUMMY DD. RDJFCB uses the stable synthetic name `NULLFILE` and reports persistent effective record attributes after a successful OPEN.

Operand	PATH	AWS tape	DATA	DUMMY	Purpose
`PATH=`	Yes	Yes	—	—	Host file/AWS image path
`RECFM=`	Yes	Yes	Yes	Yes	Record format
`LRECL=`	Yes	Yes	Yes	Yes	Logical record length
`BLKSIZE=`	Yes	Yes	Yes	Yes	Block size; maximum DD value 32760
`TRIM=YES` / `TRIM=NO`	Yes	Ignored	—	—	Trim trailing blanks on fixed PATH OUTPUT; default `NO`
`DLM=cc`	—	—	Yes	—	Optional two-character DATA terminator

Operand	PATH	AWS tape	DATA	DUMMY	Purpose
`UNIT=TAPE`	—	Yes	—	—	Select AWS tape allocation
`DSN=`	—	Yes	—	—	Tape data-set name
`VOL=SER=`	—	Yes	—	—	Volume serial
`LABEL=`	—	Yes	—	—	SL/NL/BLP and label-file selection
`EXPDT=`	—	Yes	—	—	Expiration date (`YYDDD` or `YYYY/DDD`)
`RETPD=`	—	Yes	—	—	Retention period; mutually exclusive with EXPDT
`EXPCHK=BYPASS`	—	Yes	—	—	Bypass expiration checking where supported

A tape allocation includes `UNIT=TAPE` and the required tape identity information, for example:

```
#PRAGMA DD TAPEIN,UNIT=TAPE,PATH='./input.aws',+
 DSN=MY.INPUT.FILE,VOL=SER=TAPE01,LABEL=(1,SL)
```

If a DDNAME is defined again, the later DD **completely replaces** the earlier definition; fields are not merged between DD statements.

OPEN distinguishes **supplied/allocation attributes** from **generic defaults**. Program DCB values have precedence over DD/allocation values. DATA is special because omitted record attributes are intrinsic allocation values (`FB/80/3280`), so they exist before any OPENEXIT. Existing standard-labeled tape can additionally supply RECFM/LRECL/BLKSIZE from HDR2 when neither the program DCB nor the DD explicitly supplies them. PATH, DUMMY, and new/unlabeled output tape do not manufacture omitted record attributes during this supplied-attribute merge.

When OPENEXIT is installed, it runs after that supplied/allocation/label merge and before generic defaults. Thus the exit can observe zero for still-unsupplied PATH, DUMMY, or new-tape fields, while DATA already exposes its allocation defaults and an existing labeled tape can expose HDR2 values. After normal exit return, TSDSUTIL supplies generic defaults only to fields still zero and then performs final validation. Without OPENEXIT the same two stages occur consecutively and produce the normal effective DCB.

For Linux host-file output, `TRIM=YES` affects only fixed-format records (`F`, `FA`, `FM`, `FB`, `FBA`, and `FBM`). TSDSUTIL still treats the record internally as the full LRECL; immediately before writing the host text line, trailing EBCDIC blank bytes (`X'40'`) are omitted. Leading and embedded blanks are preserved, and an all-blank fixed record is written as an empty line. `TRIM=NO` is the default and preserves the traditional exact-LRECL host output. The option is ignored for INPUT, variable/undefined formats, and AWS tape. Fixed-format Linux INPUT continues to pad short host lines with blanks to LRECL.

Chapter 6 covers host-file DCB processing and Chapter 7 covers AWS tape semantics.

9.7 Linux Configuration Files and Precedence

The Linux driver reads these optional files in order:

```
/etc/tsdsutil.conf
$HOME/.config/tsdsutil.conf
```

They contain `#PRAGMA` statements, blank lines, and `*` comments; they are not general TSDSUTIL source files. The distribution includes `etc/tsdsutil.conf`, a compact commented template for the system-wide file. `make install` copies that template to `/etc/tsdsutil.conf` only when the destination does not already exist. Configuration pragmas use the same trailing-`+` continuation convention as source pragmas: the continuation record omits `#PRAGMA`, leading whitespace is removed, and no punctuation is inserted automatically. Blank or comment-only records are errors while a configuration continuation is pending. The user file can replace settings established by the system file. Source pragmas then establish program requirements, while applicable command-line options provide run-specific overrides.

A practical division is:

Location	Best use
`/etc/tsdsutil.conf`	Installation defaults
`$HOME/.config/tsdsutil.conf`	Programmer preferences
Source pragma preamble	Requirements belonging to the program
Command line	One-run overrides

For conditional definitions, use the explicit precedence shown in §9.3. Chapter 12 documents command-line controls.

9.8 Date/Time Zone and DST Configuration

STCK/STCKE values are UTC. Civil date/time services use the configured local-time rules. Defaults are UTC with DST disabled:

```
#PRAGMA TIMEZONE_OFFSET=+00:00
#PRAGMA NO_DST
#PRAGMA DST_OFFSET=+01:00
```

Pragma	Meaning
`TIMEZONE_OFFSET`	Base UTC-to-local standard-time offset; default `+00:00`
`DST` / `NO_DST`	Enable or disable daylight-saving adjustment; default disabled
`DST_OFFSET`	Extra offset while DST is active; default `+01:00`
`DST_BEGIN`	Local civil-time rule for entering DST
`DST_END`	Local civil-time rule for leaving DST

`TIMEZONE_OFFSET` and `DST_OFFSET` use the value syntax `+HH:MM` or `-HH:MM`, where HH is 00–23 and MM is 00–59. `TIMEZONE_OFFSET` is the base offset added to UTC to obtain local **standard** time: positive values are east/ahead of UTC and negative values are west/behind UTC. `DST_OFFSET` is not an absolute time-zone offset; it is an additional adjustment added to the standard-time offset only while DST is active. For example, `TIMEZONE_OFFSET=-06:00` with `DST_OFFSET=+01:00` means UTC-06:00 in standard time and UTC-05:00 while DST is active.

Transition syntax is:

```
#PRAGMA DST_BEGIN MONTH=n,WEEK=n,DAY=ddd,TIME=HH:MM
#PRAGMA DST_END   MONTH=n,WEEK=n,DAY=ddd,TIME=HH:MM
```

`MONTH` is 1–12, `WEEK` is 1–4, and `DAY` is `SUN` through `SAT`. When DST is enabled, defaults are the current U.S.-style rules: second Sunday in March at 02:00 and first Sunday in November at 02:00.

The begin transition uses the standard UTC offset; the end transition uses standard offset plus `DST_OFFSET`. These configured rules are applied uniformly across TSDSUTIL's 1900–2899 civil-date range; TSDSUTIL does not maintain historical timezone-rule data. Local-to-timestamp conversion returns RC 28 for a nonexistent spring-transition time and RC 32 for an ambiguous fall-transition time.

9.9 Recoverable Runtime RC Messages

The date/time macro family can report supported recoverable failures in the runtime listing while still returning the documented R15 value:

```
#PRAGMA RUNTIME_RC_MESSAGES
#PRAGMA NO_RUNTIME_RC_MESSAGES
```

Suppressing these messages changes neither the return code nor TRACE/output semantics. Ordinary statuses such as tape EOF are not controlled by this option.

Chapter 10 — Execution, Diagnostics, TRACE, and Debugging

Applies to: TSDSUTIL Version 1.0.1

TSDSUTIL separates assembly diagnostics from runtime failures and provides TRACE, assertions, formatted/raw storage displays, allocation diagnostics, execution safeguards, and performance profiling. This chapter explains how those facilities work together. Appendix E is authoritative for diagnostic codes and Appendix D for return codes and ABENDs.

10.1 Assembly and Runtime Diagnostics

Assembly diagnostics are associated with source processing and normally identify the source line involved. Runtime diagnostics arise only after successful assembly when an executing instruction or service detects an error.

Diagnostic numbers use one shared four-digit namespace; severity is reported separately. The listing's DIAGNOSTIC SUMMARY provides the final assembly summary and individual flagged statements. Chapter 11 describes its presentation.

Runtime services may report a recoverable R15 status or terminate with an ABEND, depending on the service and condition. A recoverable status is not an ABEND. On termination TSDSUTIL also closes open DCBs and reports automatic cleanup where applicable.

10.2 TRACE

TRACE records the effective operation and the state relevant to understanding it. Enable it through Chapter 9 configuration or Chapter 12 command-line controls; configured `TRACE_FILE` or `--trace-file=` selects a separate host destination. `TRACE_FILE` is accepted only in the Linux system/user configuration files. The Linux driver supplies no implicit trace file, so TRACE is effective only when a usable trace destination and writer are available.

A trace record can include:

Information	When applicable
Virtual instruction address and source line	Machine instructions and macros
Effective instruction/macro operands	All traced operations
Register before/after values	Registers read or modified by the operation
Condition code	When the operation reads or changes CC
Storage before/after spans	Storage operands with their effective lengths
Service context	DDNAME, DSN, VOLSER, path, requested/actual length, tape boundary, RC, etc.

TRACE is instruction-aware rather than a generic register dump. It accounts for special operand lengths and dependencies, including mask-selected ICM/STCM/CLM bytes, TR/TRT tables, edit-pattern-selected ED/EDMK source lengths, register-defined MVCL/CLCL spans, and relevant register pairs/comparators.

When an instruction fails because a storage operand cannot be accessed, TRACE still emits the affected operand address and `LEN`. The access result distinguishes `Invalid Virtual Address` from `Write Protected`. For failed instructions, destination operands are reported for accessibility only; TRACE does not emit a misleading post-execution `AFTER=` image when the instruction never completed.

In Version 1.0.1, the runtime storage-operand diagnostics for GET, PUT, `$READ`, and `$WRITE` use that same access classification. Their established message prefix is retained, followed by `ADDRESS=hhhhhh LEN=n` and either `Invalid Virtual Address` or `Write Protected`. This makes the runtime diagnostic and the corresponding TRACE record describe the failing span consistently.

Macro operands are reconstructed before execution changes registers or storage. Date/time TRACE records the conversion/adjustment request and destination spans; those macros preserve CC. Tape TRACE can identify physical boundary conditions as well as the shared R15 status. `$CONTROL` TRACE includes the available DD, DSN, VOL, PATH, requested control type, boundary condition, and R15 result.

`EX` is traced in terms of the effective executed target so that modified lengths and resulting state are visible.

10.3 Focused Debugging Services

Full TRACE is not always necessary. TSDSUTIL provides smaller tools for specific questions:

Service	Best use
`$REGS`	Display all 16 general registers without changing them
`$DUMP`	Inspect exact virtual-storage bytes
`$DFMT`	Interpret storage through a DSECT definition
`$ASSERT`	Turn an expected condition into an executable test

`$DUMP` is appropriate when the exact byte representation matters. `$DFMT` is preferable when a DSECT gives those bytes field meaning. `$ASSERT` is particularly useful in permanent regression programs because a failed assertion terminates with a SYSTEM EEE ABEND and a TSDS reason diagnostic rather than silently allowing a bad result to propagate. Exact macro syntax is in Appendix B.

10.4 GETMAIN/FREEMAIN Debugging

`GETMAIN_DEBUG` adds allocation/release diagnostics. `GETMAIN_FILL X'xx'` changes the initialization byte for newly allocated storage; zero is the default and a conspicuous test value can expose assumptions about uninitialized data.

FREEMAIN correctness checks do not depend on debug reporting. Invalid releases, including address/size mismatches, are diagnosed according to the service rules even when `GETMAIN_DEBUG` is off.

10.5 Execution Safeguards

TSDSUTIL detects several classes of execution failure independently:

Safeguard	Behavior
Instruction limit	Default 10,000,000; reaching it reports TSDS8028E and SYSTEM EEE ABEND
Unlimited execution	`MAX_INSTRUCTIONS=0`; also forces PERFORMANCE off
Direct self-branch	Most taken branches directly to their own instruction ABEND; BCT/BCTR/BXH/BXLE are exempt because their register update can make progress
Storage protection	Rejects writes to protected virtual regions such as LOWMEM/CODE
Address/length validation	Rejects invalid spans before host memory is accessed
OPEN/tape validation	Reports service-specific allocation, label, format, positioning, and host-I/O failures

The configuration controls are in Chapter 9. Detailed tape behavior is in Chapter 7.

10.6 Execution Statistics and Performance Profiling

Normal execution statistics summarize program activity. V92 PERFORMANCE adds a more detailed end-of-run profile when enabled.

The profile includes overall execution information plus storage, I/O, instruction, and service activity. Instruction/service profiles count operations in the normal execution loop; TRACE remains independent, so measured execution time includes any TRACE overhead when both are enabled.

`MAX_INSTRUCTIONS` is a dispatch limit, not merely a count of top-level machine instructions. Executable machine instructions and executable macros each consume one dispatch; an instruction executed as an `EX` subject also passes through the same gate, so `EX` and its executed subject each consume a dispatch. `MAX_INSTRUCTIONS=0` disables PERFORMANCE to retain the lean unlimited-execution path.

The direct-self-branch safeguard is intentionally narrow. Taken self-targeting BCT, BCTR, BXH, and BXLE instructions are exempt because their register updates may eventually end the loop; if they do not, the normal instruction-dispatch limit remains the safeguard.

10.7 Completion and ABEND Reporting

Normal program completion and abnormal termination are distinct. `$EOJ` provides the normal program completion mechanism. ABEND retains SYSTEM/USER classification, ABEND code, and reason information while the Linux driver returns the configured host `ABEND_STATUS`.

Do not infer a TSDSUTIL ABEND solely from a nonzero service R15 value. Appendix D defines the authoritative completion, return-code, and ABEND rules.

10.8 Practical Debugging Workflow

A useful sequence is:

1. **Assemble without execution** when source, symbol, COPY/INCLUDE, or linkage problems are suspected; review the listing and XREF.
2. **Run with the normal execution limit** and inspect the first runtime diagnostic rather than secondary effects.
3. **Use `$REGS`, `$DUMP`, or `$DFMT`** when the problem is localized.
4. **Enable TRACE** when control flow, register evolution, storage modification, or service interaction must be reconstructed.
5. **Add `$ASSERT` checks** once an expected invariant is known so the same defect cannot silently recur.
6. **Use PERFORMANCE** only when execution behavior is correct and the question has become where time or activity is concentrated.

This combination makes TSDSUTIL useful not only for one-time data manipulation but also as a repeatable test environment for assembler-style algorithms and utility programs.

Chapter 11 — Listings and Cross-Reference

Applies to: TSDSUTIL Version 1.0.1

The assembly listing is the permanent record of how TSDSUTIL interpreted a source program. XREF adds symbol-definition and reference information. This chapter defines their presentation; Chapter 10 covers runtime TRACE and debugging.

11.1 Requesting and Formatting a Listing

The Linux driver writes a listing when `--listing=path` is supplied. `LISTING_WIDTH` controls its width from 132 through 255 columns; the default is 132. `CODE_LISTING` controls generated code/instruction detail, while `RUNTIME_LISTING` controls whether runtime program output is also copied into the listing.

Item	Listing behavior
Main source	Assigned listing line numbers
Continuations	Presented as part of the continued logical statement
`#COPY` source	Remains in the current translation unit
`#INCLUDE` source	Identified with its separate translation unit where needed
DATA symbols	Show their virtual DATA addresses
CODE statements	Use virtual CODE addresses beginning in the fixed CODE region
Diagnostics	Shown with the affected source and summarized at the end
Runtime output	Included when `RUNTIME_LISTING` is enabled
Completion/statistics	Appended for executed programs as applicable

The exact virtual-storage map is in Appendix F.

11.2 Diagnostic Summary

The DIAGNOSTIC SUMMARY begins with the count of statements flagged by diagnostics:

```
No Statements Flagged
```

or the corresponding count when statements were flagged. Individual `Line n TSDS####S ...` entries follow it.

This gives both a quick assembly result and a stable location for reviewing all assembly diagnostics without searching the complete listing.

11.3 Cross-Reference (XREF)

Enable XREF with `#PRAGMA XREF`; `XREF_LONG` includes information normally omitted from the shorter form, including unreferenced symbols where applicable.

XREF includes all symbols relevant to assembled expressions, including the built-in register symbols R0–R15.

XREF information	Meaning
Definition line	Main listing line where the symbol is defined
Reference lines	Parsed source expressions that reference the symbol
Undefined symbol	Referenced but not resolved by assembly
Unreferenced symbol	Defined but unused; shown in the long form where applicable
DSECT field	Definition/reference information for DSECT symbols
ENTRY/VCON	Cross-TU linkage definitions/references
TU identifier	Added only when duplicate spellings in different translation units require disambiguation

XREF reports semantic references from parsed expressions rather than simple textual occurrences. Literals and generated/internal information are represented according to their assembler role rather than by searching source text for matching character strings.

11.4 Using XREF to Diagnose Programs

XREF is especially useful for a few recurring problems:

Symptom	What to inspect
Unexpected undefined symbol	Spelling, TU ownership, ENTRY/VCON linkage, or deferred expression
Apparently duplicate names	Translation-unit identifiers and definition lines
Definition never used	Long-form unreferenced-symbol information
Wrong register alias	R0–R15 references and user EQU aliases
Unexpected DSECT-field use	Reference lines associated with the field

For deferred expressions, the final unresolved-symbol diagnostic is authoritative; XREF complements it by showing where the names were referenced.

11.5 Listing, TRACE, and Storage Displays

These facilities answer different questions:

Facility	Primary question
Listing	What did the assembler build, and what diagnostics occurred?
XREF	Where was each symbol defined and referenced?
TRACE	What happened while the program executed?
`$REGS`	What are the current register values?
`$DUMP`	What exact bytes are in this storage span?
`$DFMT`	What do these bytes mean under this DSECT?

A productive development configuration normally keeps a listing, enables XREF when symbol/linkage work is active, and adds TRACE only when runtime behavior requires it. Because the listing records source, diagnostics, addresses, XREF, and optionally runtime information, it can also serve as useful program documentation for a completed utility.

Chapter 12 — Linux Command-Line Interface

Applies to: TSDSUTIL Version 1.0.1

The Linux `tsdsutil` driver reads primary source from standard input by default, or from a named script selected with `--script`/`-s`. Command-line options select run-specific execution, listing, TRACE, conditional-definition, search-path, and simple DD settings.

```
tsdsutil [-V|-v]
         [--script=path|-s path]
         [--execute|--no-execute]
         [--listing=path]
         [--parm=value]
         [--define NAME[=VALUE] ...]
         [--max-instructions=n]
         [--performance|--no-performance]
         [--trace|--trace-active|--no-trace]
         [--trace-file=path]
         [-Lpath ...]
         [--ddDDNAME=path ...]
```

12.1 Command-Line Options

Option	Meaning
`--help`, `-h`	Display usage and exit
`-V`, `-v`	Display the TSDSUTIL version and exit successfully before normal startup processing
`--script=path`, `--script path`, `-s path`	Read the primary source from a named script instead of standard input
`--execute`	Execute after successful assembly
`--no-execute`	Assemble without executing
`--listing=path`	Write the assembly listing to `path`
`--parm=value`	Supply startup PARM; maximum 100 bytes
`--define NAME`	Define a CLI-precedence conditional name
`--define NAME=VALUE`	Define a CLI-precedence name and value
`--max-instructions=n`	Override execution limit; non-negative decimal, `0` = unlimited
`--performance`	Enable detailed end-of-run performance profiling
`--no-performance`	Disable performance profiling
`--trace`	Enable permitted TRACE mode
`--trace-active`	Select active TRACE mode
`--no-trace`	Disable TRACE
`--trace-file=path`	Select TRACE output file
`-Lpath`, `-L path`	Add a COPY/INCLUDE search directory
`--ddDDNAME=path`	Associate a host path with a DDNAME; an empty path removes/masks that command-line association

`--listing=` and `--trace-file=` require nonempty paths. Only one `--script`/`-s` may be specified; when present it overrides/ignores standard input. Multiple `--define`, `-L`, and `--dd` options may be supplied. The Linux driver accepts up to 128 `--define` operands, 64 `-L` search paths, and 64 command-line DD associations.

`--ddDDNAME=` with an empty path is a deliberate removal form. It masks a lower-level source/configuration DD path for that run so the DDNAME is treated as unallocated unless another applicable host association remains.

`--define` also accepts `--define=NAME` and `--define=NAME=VALUE`. The definition itself cannot contain whitespace. Conditional-definition precedence is described in Chapter 8.

`--max-instructions=0` requests unlimited execution and forces PERFORMANCE off even if `--performance` is also present. Linux TRACE has no implicit file destination: `--trace` or `--trace-active` alone does not create trace output. A usable `--trace-file=` (or configured `TRACE_FILE`) is required; inability to open that file is non-fatal and leaves effective tracing off. Execution limits, TRACE, and profiling behavior are described in Chapter 10.

12.2 Source, Data, and Output Streams

Primary TSDSUTIL source is standard input unless a named script is selected:

```
./tsdsutil <program.tsds
./tsdsutil --script=program.tsds
./tsdsutil --script program.tsds
./tsdsutil -s program.tsds
```

For a named script, TSDSUTIL first tries the path exactly as entered. If that path is not found and the final filename component has no extension, it retries with lowercase `.tsds` appended. Thus `-s rand` tries `rand` first and then `rand.tsds`. Errors other than "not found" do not trigger the fallback. The actual opened filename supplies the script stem used by configured output templates.

Because standard input carries source, program data input is supplied through DD allocation rather than simultaneously using the host standard-input stream:

```
./tsdsutil --ddSYSIN=./data.txt <program.tsds
```

Normal program/messages output and diagnostics can be separated by the shell:

```
./tsdsutil <program.tsds >program.out 2>program.err
```

Listings and TRACE can each use separate files:

```
./tsdsutil --listing=program.lst --trace --trace-file=program.trc <program.tsds
```

The shell processes quoting before TSDSUTIL sees arguments. Quote values containing spaces or shell-significant characters, for example `--parm="MONTHLY REPORT"`.

12.3 Command-Line DD Associations

The simple command-line form associates a path with a DDNAME:

```
./tsdsutil --ddSYSIN=./input.dat --ddREPORT=./report.dat <program.tsds
```

Use source/configuration `#PRAGMA DD` when allocation attributes such as `RECFM`, `LRECL`, `BLKSIZE`, or AWS tape parameters must also be specified. Chapters 6, 7, and 9 describe those allocations.

12.4 Configuration and Overrides

Before source assembly, the Linux driver reads the optional system and user configuration files described in Chapter 9, processes command-line definitions/associations, and applies explicit run controls.

The configuration files are `/etc/tsdsutil.conf` for system defaults and `$HOME/.config/tsdsutil.conf` for per-user preferences. The user file can override system settings; source pragmas and applicable command-line options then provide progressively more specific run controls.

Command-line execution, TRACE, performance, and instruction-limit selections are run-specific overrides according to their option rules. Conditional definitions use the separate hierarchy documented in Chapter 8:

```
BUILTIN < SYSTEM < PRAGMA < CLI < SOURCE
```

12.5 Host Errors and Process Status

Malformed driver arguments are rejected before assembly. Examples include unknown options, empty required paths, invalid definitions, PARM over 100 bytes, malformed instruction limits, missing `-L` directories, and invalid/excessive DD associations.

The Linux process status is not necessarily the program's R15 value. Core host-level statuses include:

Status	Meaning
0	`TSDS_STATUS_OK`
8	`TSDS_STATUS_ERROR`
16	`TSDS_STATUS_SEVERE`

A normally executed program can return its completion code through `$EOJ`. ABEND uses the configured `ABEND_STATUS` (default 100) while preserving the TSDSUTIL ABEND classification/code/reason in diagnostics. Appendix D is authoritative for completion and ABEND behavior.

12.6 Common Invocations

Goal	Example
Normal run	`./tsdsutil <program.tsds`
Listing	`./tsdsutil --listing=program.lst <program.tsds`
Assembly only	`./tsdsutil --no-execute --listing=program.lst <program.tsds`
Conditional test	`./tsdsutil --define TEST --define LEVEL=3 <program.tsds`
Trace a run	`./tsdsutil --trace-active --trace-file=program.trc --listing=program.lst <program.tsds`
External data sets	`./tsdsutil --ddSYSIN=./input.dat --ddREPORT=./report.dat <program.tsds`
Source libraries	`./tsdsutil -L./copy -L./modules <program.tsds`
Lower development limit	`./tsdsutil --max-instructions=1000000 --listing=program.lst <program.tsds`

These controls allow the run environment to change diagnostics, source search, external paths, and execution policy without editing the program itself.

12.8 Linux Installation Targets

TSCLIB must be installed or otherwise available before building TSDSUTIL. TSCLIB is a separate prerequisite and is not included in the TSDSUTIL distribution. The Makefile searches `../tsclib`, `$HOME/tsclib`, `/usr/local`, and `/usr`, or an explicit `TSC_PATH=/path/to/tsclib`.

Build TSDSUTIL as a normal user first:

```
make
```

The supplied Makefile then provides conventional installation targets. By default:

```
sudo make install
```

installs the executable as `/usr/bin/tsdsutil`. It also creates `/etc/tsdsutil.conf` from the distribution's `etc/tsdsutil.conf` template only when the system file does not already exist. A pre-existing configuration file is preserved unchanged.

The installation paths are parameterized:

```
PREFIX=/usr
BINDIR=$(PREFIX)/bin
SYSCONFDIR=/etc
DESTDIR=
```

`DESTDIR` supports staged/package installations without changing the paths that will exist in the installed image.

`make uninstall` removes the installed executable but deliberately leaves `/etc/tsdsutil.conf` in place because it may contain administrator changes. `make purge` removes both the executable and the configuration file and should therefore be used only when that configuration is no longer wanted.

Appendix A — Machine Instruction Reference

Applies to: TSDSUTIL Version 1.0.1

This appendix lists every System/370-style machine-instruction mnemonic accepted by the current instruction table. TSDSUTIL follows System/370 semantics unless a difference is documented here; this is a support and syntax reference rather than a replacement for IBM's instruction descriptions.

TSDSUTIL macros such as `$PRINT`, OPEN, GETMAIN, and `$READ` are not machine instructions and are documented separately.

A.1 General Rules

Mnemonics and symbols are case-insensitive. Registers are R0 through R15, and a register operand can be an expression resolving to 0–15, so an EQU name can serve as a register name.

Storage can be addressed by a label or by normal displacement/register notation. A label is already a virtual address; no code base register is required. `USING` applies only to DSECTs.

TSDSUTIL uses 24-bit virtual addresses. LOWMEM 000000–000FFF is read-only, DATA begins at 001000, and CODE occupies F00000–FFFFFF with executable addresses beginning at F00002. CODE is not general data storage.

Runtime operands need not be naturally aligned. Address-range and storage-protection checks still apply.

A.2 Instruction Formats

Format	General form
RR	`R1,R2`
RX	`R1,D2(X2,B2)`
RS	`R1,R3,D2(B2)`
SI	`D1(B1),I2`
S	`D2(B2)`
I	`I2`
SS1	`D1(L,B1),D2(B2)`
SS2	`D1(L1,B1),D2(L2,B2)`
SS3	Instruction-specific SS form

A.3 Complete Instruction List

Mnemonic	Format	Source form	TSDSUTIL note
`A`	RX	`R1,D2(X2,B2)`	
`AH`	RX	`R1,D2(X2,B2)`	
`AL`	RX	`R1,D2(X2,B2)`	
`AR`	RR	`R1,R2`	
`BAL`	RX	`R1,D2(X2,B2)`	
`SPM`	RR	`R1`	
`BALR`	RR	`R1,R2`	
`BAS`	RX	`R1,D2(X2,B2)`	
`SVC`	I	`I2`	
`BASR`	RR	`R1,R2`	
`BC`	RX	`M1,D2(X2,B2)`	Normal CC mask mapping is 8,4,2,1 for CC 0,1,2,3.
`BCR`	RR	`M1,R2`	Normal CC mask mapping is 8,4,2,1 for CC 0,1,2,3.
`BCT`	RX	`R1,D2(X2,B2)`	
`BCTR`	RR	`R1,R2`	
`BXH`	RS	`R1,R3,D2(B2)`	
`BXLE`	RS	`R1,R3,D2(B2)`	
`C`	RX	`R1,D2(X2,B2)`	

Mnemonic	Format	Source form	TSDSUTIL note
CH	RX	R1,D2(X2,B2)	
CL	RX	R1,D2(X2,B2)	
CLC	SS1	D1(L,B1),D2(B2)	
CLI	SI	D1(B1),I2	
CR	RR	R1,R2	
CLR	RR	R1,R2	
CDS	RS	R1,R3,D2(B2)	
CLM	RS	R1,M3,D2(B2)	Storage length is the number of bytes selected by the mask.
CS	RS	R1,R3,D2(B2)	
CVB	RX	R1,D2(X2,B2)	
CVD	RX	R1,D2(X2,B2)	
D	RX	R1,D2(X2,B2)	
DR	RR	R1,R2	
ED	SS1	D1(L,B1),D2(B2)	Source consumption is derived from the pre-execution edit pattern.
EDMK	SS1	D1(L,B1),D2(B2)	Source consumption is derived from the pre-execution edit pattern.
EX	RX	R1,D2(X2,B2)	Executes the effective target from TSDSUTIL's compiled instruction representation; supported SI targets are included.
IC	RX	R1,D2(X2,B2)	
ICM	RS	R1,M3,D2(B2)	Storage length is the number of bytes selected by the mask.
L	RX	R1,D2(X2,B2)	
LA	RX	R1,D2(X2,B2)	Effective address follows TSDSUTIL 24-bit virtual-address behavior.
LH	RX	R1,D2(X2,B2)	
LM	RS	R1,R3,D2(B2)	
MVCL	RR	R1,R2	Addresses and lengths are register-defined.
CLCL	RR	R1,R2	Addresses and lengths are register-defined.
LPR	RR	R1,R2	
LNR	RR	R1,R2	
LTR	RR	R1,R2	
LCR	RR	R1,R2	
LR	RR	R1,R2	
M	RX	R1,D2(X2,B2)	
MH	RX	R1,D2(X2,B2)	
MR	RR	R1,R2	
MVN	SS1	D1(L,B1),D2(B2)	
MC	SI	D1(B1),I2	
MVC	SS1	D1(L,B1),D2(B2)	Movement is left-to-right, including overlapping operands.
MVI	SI	D1(B1),I2	
N	RX	R1,D2(X2,B2)	
MVZ	SS1	D1(L,B1),D2(B2)	
NC	SS1	D1(L,B1),D2(B2)	
TS	S	D2(B2)	

Mnemonic	Format	Source form	TSDSUTIL note
NI	SI	D1(B1),I2	
NR	RR	R1,R2	
O	RX	R1,D2(X2,B2)	
OC	SS1	D1(L,B1),D2(B2)	
OI	SI	D1(B1),I2	
OR	RR	R1,R2	
MVCIN	SS1	D1(L,B1),D2(B2)	
MVO	SS2	D1(L1,B1),D2(L2,B2)	
PACK	SS2	D1(L1,B1),D2(L2,B2)	
ZAP	SS2	D1(L1,B1),D2(L2,B2)	
CP	SS2	D1(L1,B1),D2(L2,B2)	
AP	SS2	D1(L1,B1),D2(L2,B2)	
SP	SS2	D1(L1,B1),D2(L2,B2)	
MP	SS2	D1(L1,B1),D2(L2,B2)	
DP	SS2	D1(L1,B1),D2(L2,B2)	
S	RX	R1,D2(X2,B2)	
SH	RX	R1,D2(X2,B2)	
SL	RX	R1,D2(X2,B2)	
SLA	RS	R1,D2(B2)	
SLDA	RS	R1,D2(B2)	
SLDL	RS	R1,D2(B2)	
SLL	RS	R1,D2(B2)	
ALR	RR	R1,R2	
SLR	RR	R1,R2	
SR	RR	R1,R2	
SRP	SS3	D1(L1,B1),D2(B2),I3	
SRA	RS	R1,D2(B2)	
SRDA	RS	R1,D2(B2)	
SRDL	RS	R1,D2(B2)	
SRL	RS	R1,D2(B2)	
ST	RX	R1,D2(X2,B2)	
STC	RX	R1,D2(X2,B2)	
STCK STCKE	S S	D2(B2) D2(B2)	
STCM	RS	R1,M3,D2(B2)	Storage length is the number of bytes selected by the mask.
STH	RX	R1,D2(X2,B2)	
STM	RS	R1,R3,D2(B2)	
TM	SI	D1(B1),I2	
TR	SS1	D1(L,B1),D2(B2)	Operand 2 is a 256-byte table.
TRT	SS1	D1(L,B1),D2(B2)	Operand 2 is a 256-byte table.
UNPK	SS2	D1(L1,B1),D2(L2,B2)	
X	RX	R1,D2(X2,B2)	
XC	SS1	D1(L,B1),D2(B2)	
XI	SI	D1(B1),I2	
XR	RR	R1,R2	
B	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
NOP	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.

Mnemonic	Format	Source form	TSDSUTIL note
NOPR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BE	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BER	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BNE	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BNER	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BH	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BHR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BL	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BLR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BNH	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BNHR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BNL	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BNLR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BP	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BPR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BM	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BMR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BZ	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BZR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.

Mnemonic	Format	Source form	TSDSUTIL note
BO	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BOR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BNP	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BNPR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BNM	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BNMR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BNZ	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BNZR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.
BNO	RX	R1,D2(X2,B2)	Extended branch alias; normalized internally to BC/BCR.
BNOR	RR	R1,R2	Extended branch alias; normalized internally to BC/BCR.

A.4 Extended Branch Mnemonics

TSDSUTIL accepts IBM extended branch mnemonics as assembler aliases. They do not create separate runtime opcodes; they are normalized to BC or BCR with the implied mask.

Storage-target aliases:

`B NOP BE BNE BH BL BNH BNL BP BM BZ BO BNP BNM BNZ BNO`

Register-target aliases:

`BR NOPR BER BNER BHR BLR BNHR BNLR BPR BMR BZR BOR BNPR BNMR BNZR BNOR`

For BC/BCR, mask bits 8, 4, 2, and 1 correspond respectively to CC 0, 1, 2, and 3.

A.5 Address Operands

A storage address can commonly be written symbolically:

```
L     R3,COUNT
MVC   OUTPUT(80),INPUT
```

or with explicit registers:

```
L     R3,0(R4)
MVC   0(80,R5),0(R6)
```

RX operands use `D2(X2,B2)`; many RS, SI, S, and SS operands use `D2(B2)`. The register fields contribute to the effective address according to the instruction format.

A zero base or index field can architecturally mean “no register contribution”; that is distinct from using the contents of R0 as an ordinary register operand.

A.6 EX

EX implements the System/370 concept of modifying and executing another instruction, but TSDSUTIL does so against its internal compiled instruction representation rather than writable machine-code bytes.

The low-order eight bits of the register specified by EX are ORed into bits 8-15, the second byte, of the subject instruction. The modified instruction instance is then executed without changing the instruction stored in CODE.

EX supports the implemented architecturally valid subject instructions, including the SI instructions:

`CLI  MC  MVI  NI  OI  TM  XI`

For these SI instructions the second byte is the immediate operand, so EX modifies that immediate value.

`TS` is a valid EX subject. Because TS ignores bits 8-15, the EX modifier has no effect on the TS operation.

`STCK` has the two-byte opcode `B205`. An EX modifier of zero leaves the opcode unchanged. A nonzero modifier is ORed into `05`; if this produces a `B2xx` opcode not implemented by TSDSUTIL, the effective instruction is reported as unsupported.

EX cannot itself be the subject of EX; attempting to do so causes an execute exception.

TRACE reports the effective executed target. This is particularly useful when EX changes an SS length or an SI immediate operand.

CODE itself remains protected from ordinary data reads and writes.

STCK

`STCK D2(B2)` stores the current eight-byte IBM TOD clock value.

- Linux builds obtain the value through TSCLIB `mvsSTCKNow()` and return CC=0 on success.
- Native GNU-compatible s390/s390x builds execute the hardware `STCK` instruction directly and preserve its condition code.

The Linux and native paths use the same architected eight-byte STCK/TOD representation.

STCKE

`STCKE D2(B2)` stores the current sixteen-byte IBM extended TOD clock value. Its opcode is `B278`. Byte 0 is the TOD epoch index, bytes 1-13 contain TOD clock bits 0-103, and bytes 14-15 contain the TOD programmable field.

- Linux builds synthesize the architected extended TOD value from the host UTC clock and return CC=0 on success.
- Native GNU-compatible s390/s390x builds execute the hardware `STCKE` instruction directly and preserve its condition code.
- TSDSUTIL writes zero in the TOD programmable field on Linux.

A.7 Storage-to-Storage Details

MVC follows left-to-right semantics, which matters for overlapping operands.

ICM, STCM, and CLM use only the number of storage bytes selected by their masks.

TR and TRT use a 256-byte second-operand table.

ED and EDMK determine source use from the edit pattern as it exists before execution.

MVCL and CLCL obtain addresses and lengths from their designated register pairs.

These same operand spans are reflected by TRACE.

A.8 Addressing and Alignment

LA produces a 24-bit virtual effective address. General registers remain 32-bit.

CODE addresses can be used for branch/linkage but are not general data-storage addresses.

DS/DC allocation follows the alignment rules documented in Chapter 3, but the execution engine permits an instruction to reference an unaligned runtime address. It validates the address and range rather than rejecting the operation solely because of alignment.

A.9 Scope of Support

Only the mnemonics listed in A.3 are supported machine instructions. An instruction from another IBM architecture or System/370 facility should not be assumed to be implemented unless it appears in the table.

Assembler directives, data-definition operations, and TSDSUTIL macros are separate language facilities and are covered elsewhere in this manual.

Appendix B — Macro Instruction Reference

Applies to: TSDSUTIL Version 1.0.1

This appendix is the quick-reference catalog for the executable macro instructions supported by the current release. The catalog is checked against the current opcode table and contains **39 built-in macro instructions**.

Macros are assembled into executable TSDSUTIL CODE instructions, but they are not System/370 machine instructions. They provide services such as program termination, diagnostics, storage management, data-set I/O, console-style messages, DCB interrogation, and physical tape I/O.

B.1 General Macro Syntax

A macro can have an optional statement label:

```
[label] macro operands
```

Macro operands are parsed according to the macro operand rules described in Chapter 5. Parentheses and commas are structural; a null operand and an empty list are distinct.

Address operands generally accept a normal address expression or a register-derived form where documented. Register-derived addresses are normalized to the 24-bit TSDSUTIL virtual address space.

Unless a macro explicitly documents condition-code effects, it leaves CC unchanged.

B.2 Quick Reference

Macro	Primary purpose
`$ASSERT`	Runtime assertion
`$CVTDATE`	Convert JDATE/GDATE/STIME/STCK/STCKE
`$DAYADJ`	Add/subtract calendar days from a JDATE
`$DAYDIFF`	Return calendar-day difference
`$DFMT`	Format storage using a DSECT
`$DUMP`	Display raw storage
`$EOJ`	End program execution
`$EXPDATE`	Expand JDATE into EBCDIC calendar fields
`$FMTDATE`	Format JDATE/TIME as EBCDIC text
`$LINENO`	Return current listing line
`$MODDCB`	Modify selected fields of a closed DCB
`$PRINT`	Print text
`$RAND`	Obtain pseudo-random value
`$CONTROL`	Position an open physical BLP AWS tape
`$READ`	Read one physical tape block
`$REGS`	Display all general registers
`$RESTORE`	Restore saved registers
`$RETURN`	Restore registers and return
`$SAVE`	Save all general registers
`$SHOWDCB`	Return selected DCB attributes
`$SLEEP`	Delay execution
`$SRAND`	Seed pseudo-random generator
`$TESTDCB`	Test one DCB attribute or OPEN state
`$TIMEUSD`	Return CPU and elapsed time
`$TRACE`	Runtime trace control
`$TIMEADJ`	Add elapsed seconds to STIME
`$VALDATE`	Validate JDATE/GDATE
`$VALTIME`	Validate IBM TIME value
`$WRITE`	Write physical tape block or tape mark
`ABEND`	Abnormally terminate execution
`CLOSE`	Close one or more DCBs
`FREEMAIN`	Release dynamic storage
`GET`	Read one logical record
`GETMAIN`	Allocate dynamic storage

Macro	Primary purpose
OPEN	Open one or more DCBs
PUT	Write one logical record
TIME	Return current IBM-style date/time
WTO	Write host/operator message
WTOR	Write message and obtain reply

B.3 $ASSERT

```
$ASSERT left,operator,right
```

Supported operators are:

```
EQ NE GT GE LT LE
```

The left operand can be a register (including an EQU alias whose value is 0–15) or memory. Memory assertions compare the number of bytes implied by a C, X, P, or B literal on the right. Register assertions compare 4-byte unsigned values; a typed right-side literal must therefore be exactly four bytes. Runtime-register and address forms are not valid on the right.

A failed assertion issues a severe TSDSUTIL diagnostic and terminates with a SYSTEM=EEE ABEND whose REASON identifies the assertion diagnostic.

A successful assertion has no program-visible side effects.

Examples:

```
$ASSERT R15,EQ,0
$ASSERT BUFFER,EQ,C'HELLO'
$ASSERT FIELD,GE,P'123'
```

B.4 $DFMT

```
$DFMT dsect,address
```

Formats storage at the explicitly supplied address according to the named DSECT.

USING is not used to select the displayed address.

Named fields are displayed with offsets, effective addresses, declared types, interpreted values, and useful hexadecimal representations. Unnamed fields and EQU definitions are omitted. Packed-decimal fields show both interpreted and hexadecimal values and flag invalid packed data.

An ambiguous overlay is reported and formatting stops rather than continuing with a misleading layout.

B.5 $DUMP

```
$DUMP area-address,area-length
```

Displays raw virtual storage.

The output includes address, hexadecimal bytes, and printable interpretations. $DUMP does not interpret storage through a DSECT.

No registers or CC are changed.

If the requested range reaches invalid, unreadable, unassigned, or CODE storage, the dump is stopped and a runtime warning is issued.

B.6 $EOJ

```
$EOJ value
$EOJ (register)
$EOJ ,
```

Ends normal execution.

If no value is supplied, the current R15 is used as the completion code. Therefore:

```
$EOJ  ,
```

and:

```
$EOJ  (R15)
```

are equivalent.

On Linux, the host completion mapping uses the low-order 8 bits of the logical completion code.

Open DCBs are automatically closed during termination.

B.7 $LINENO

```
$LINENO ,
```

Returns the current source/listing line number in R15.

This is useful for program-generated diagnostics and reports that need to identify the executing source statement.

B.8 $PRINT

Quoted-string form:

```
$PRINT "text"
```

Storage form:

```
$PRINT text-address,text-length
```

Writes EBCDIC text followed by an implied newline. Quoted text is converted to EBCDIC at assembly time; the Linux host layer converts EBCDIC output to native text.

The storage form is useful for dynamically constructed messages.

B.9 $RAND

```
$RAND ,
```

Returns the next 32-bit value from the TSDSUTIL pseudo-random number generator in R1. R15 is unchanged.

Use `$SRAND` when a repeatable or host-derived seed is required.

B.10 $READ

```
$READ dcb-address,buffer-address,maximum-length
```

Reads one **physical tape block** from a tape DCB opened for INPUT with `MACRF=R` or `RW`.

R1 receives the full physical block length.

R15 values are:

R15	Meaning
0	Data block read
4	Logical EOF for SL/NL
8	Block longer than caller buffer; prefix copied and residual discarded
12	Physical tape mark in BLP mode
16	Physical end-of-image/EOT
20	Read attempted after EOT
24	Invalid operation or state
28	Tape/host I/O error
32	Volume/data-set format error
36	Internal/resource error

A maximum length of zero is valid. The block is consumed and discarded while R1 reports its full length.

Invalid or protected destination storage ABENDs.

`$READ` is a physical interface; logical record input uses GET.

B.11 $REGS

```
$REGS ,
```

Displays all 16 general registers, four per output line.

Typical format:

```
R00 XXXXXXXX R01 XXXXXXXX R02 XXXXXXXX R03 XXXXXXXX
```

No registers or CC are changed.

B.12 $RESTORE

```
$RESTORE ,
```

Restores all 16 registers from the most recent `$SAVE` register image and pops that save frame. CC is unchanged. `$RETURN` likewise pops the frame after restoring registers (subject to its supported exclusions), then returns through the restored R14.

CC is unchanged.

A save-stack underflow ABENDs.

Use `$RETURN` when the intent is to restore registers and return through R14 in one operation.

B.13 $RETURN

No-operand form:

```
$RETURN
```

or:

```
$RETURN ,
```

restores the saved registers and branches through restored R14.

An exclusion list can preserve selected current return registers rather than restoring their saved values:

```
$RETURN (R0)
$RETURN (R1)
$RETURN (R15)
$RETURN (R0,R1)
$RETURN (R15,R0)
$RETURN (R15,R1)
```

The supported exclusions are limited to R0, R1, and R15 and the valid combinations defined by the macro.

This is the normal companion to `$SAVE`.

B.14 $SAVE

```
$SAVE ,
```

Pushes all 16 general registers onto the TSDSUTIL save stack.

The registers and CC are unchanged by the save itself.

Save-stack overflow ABENDs.

A typical subroutine is:

```
ROUTINE  $SAVE ,
         ...
         $RETURN
```

B.15 $SHOWDCB

```
$SHOWDCB DCB=dcb,FIELDS=field,AREA=address,AREALEN=bytes
```

or:

```
$SHOWDCB DCB=dcb,FIELDS=(field,...),AREA=address,AREALEN=bytes
```

Supported fields are:

Field	Returned size	Representation
LRECL	4	Fullword
BLKSIZE	4	Fullword
DDNAME	8	EBCDIC, blank padded

Fields are returned in the exact order requested. Thus:

```
FIELDS=(LRECL,BLKSIZE)
```

and:

```
FIELDS=(BLKSIZE,LRECL)
```

produce the two fullwords in opposite order.

Each field may appear only once. A duplicate field is an assembly error. The DCB and AREA operands accept ordinary addresses, displacement/base addresses, and structural runtime-register addresses where applicable.

R15 values are:

```
0  success
4  return area too small
8  invalid DCB, invalid return area, or other query error
```

Closed DCBs expose their current baseline values. While a DCB is open, the macro exposes the effective merged values established by OPEN. CLOSE restores the baseline DCB values; that baseline may include earlier `$MODDCB` changes and therefore need not equal the values originally assembled.

B.16 $SLEEP

```
$SLEEP seconds
$SLEEP (register)
```

Suspends execution for the specified whole number of seconds. An ordinary expression is evaluated at assembly time; `(register)` obtains the delay from that register at execution time.

Registers and CC are unchanged.

B.17 $SRAND

```
$SRAND (register)
```

Seeds the TSDSUTIL pseudo-random generator.

If the specified register is R0, TSDSUTIL requests a host-derived non-repeatable seed.

Any other register supplies its exact 32-bit contents as the deterministic seed, including a value of zero.

Example:

```
        L     R4,=F'12345'
        $SRAND (R4)
```

produces a repeatable random sequence.

B.18 $TESTDCB

Syntax

```
$TESTDCB DCB=dcb,RECFM=value
$TESTDCB DCB=dcb,LRECL=value
$TESTDCB DCB=dcb,BLKSIZE=value
$TESTDCB DCB=dcb,DDNAME=name
$TESTDCB DCB=dcb,DDNAME=address|(register)
$TESTDCB DCB=dcb,OPEN=YES
$TESTDCB DCB=dcb,OPEN=NO
```

Exactly one attribute or state test is performed.

On a successful test:

```
R15 = 0
CC=0  requested value/state is equal or true
CC=1  requested value/state is not equal or false
```

`OPEN=YES` tests whether the DCB is currently open. `OPEN=NO` tests whether it is currently closed.

RECFM comparisons are exact. `FB` matches only `FB`.

`LRECL` and `BLKSIZE` accept ordinary expressions or structural runtime register values. `DDNAME` may be a literal name or the address of an eight-byte EBCDIC, blank-padded value.

If the test cannot be performed, R15 is nonzero and CC is preserved.

An open DCB exposes effective merged values. A closed DCB exposes its current baseline values.

B.19 $MODDCB

Syntax

```
$MODDCB DCB=dcb,[DDNAME=name],[RECFM=value],[LRECL=value],[BLKSIZE=value],[OPENEXIT=address|0]
```

At least one modification keyword is required.

For a normally closed DCB, `$MODDCB` changes the DCB baseline for the remainder of execution. `OPENEXIT` installs a subsequent OPEN exit, and `OPENEXIT=0` disables it. A nonzero OPENEXIT value uses the normal DCB address-expression rules and is checked as executable when OPEN attempts to invoke it.

While an OPENEXIT is active, `$MODDCB` is intentionally different: it may change only the provisional effective `RECFM`, `LRECL`, and `BLKSIZE` for the DCB being opened. Those changes do not rewrite the closed-DCB baseline. An attempt inside OPENEXIT to change `DDNAME` or `OPENEXIT` causes SYSTEM EEE.

Changes are atomic for the requested fields. `LRECL` and `BLKSIZE` accept ordinary expressions or structural runtime register values. `DDNAME` may be a literal name or an address (including a structural runtime-register address) to an eight-byte EBCDIC value. Cross-field RECFM/LRECL/BLKSIZE compatibility is checked by final OPEN validation.

Return codes:

R15	Meaning
0	Modification completed
4	Valid DCB is currently open
8	One or more requested field values are invalid
12	Request cannot otherwise be processed, such as invalid DCB

CC is preserved.

B.20 $TIMEUSD

```
$TIMEUSD ,
```

Returns timing values as integer thousandths of a second:

```
R0  CPU time
R1  elapsed time
```

CC is unchanged.

Despite the historical macro name, the V67 values are milliseconds (thousandths), not microseconds.

B.21 $TRACE

```
$TRACE ...
```

`$TRACE` is the executable runtime TRACE-control macro. It is distinct from `#PRAGMA TRACE`, which establishes trace configuration before execution.

The macro is compiled as an executable TSDSUTIL macro instruction and participates in normal macro execution/TRACE accounting.

Use the TRACE facilities described in Chapter 10 for execution tracing; use `#PRAGMA TRACE`, `TRACE_ACTIVE`, configured `TRACE_FILE`, or their command-line equivalents to establish run-level tracing.

B.22 $WRITE

Data form:

```
$WRITE dcb-address,buffer-address,length
```

writes exactly one **physical tape block** through a tape DCB opened for OUTPUT with `MACRF=W` or `RW`.

Tape-mark form:

```
$WRITE dcb-address,TYPE=TM
```

writes one physical tape mark in BLP mode.

R15 uses the common physical tape status namespace:

R15	Meaning
0	Success
8	Block larger than effective BLKSIZE
20	Operation after EOT
24	Invalid operation or state
28	Tape/host I/O error
32	Volume/data-set format error
36	Internal/resource error

A data length of zero is invalid.

Invalid or protected source storage ABENDs.

`TYPE=TM` uses the same return-code classes as a physical data write.

`$WRITE` is a physical interface; logical record output uses PUT.

B.23 $CONTROL — Physical Tape Positioning

```
$CONTROL dcb-address,TYPE=BSB|BSF|FSB|FSF[,COUNT=value|(register)]
$CONTROL dcb-address,TYPE=REWIND
```

Controls the physical position of an open BLP AWS tape DCB using `MACRF=R`, `W`, or `RW`. `BSB`/`FSB` space physical blocks, `BSF`/`FSF` space tape files, and `REWIND` positions to BOT. `COUNT=` defaults to one and accepts an ordinary expression or structural runtime-register value. It is valid only for BSB, BSF, FSB, and FSF; REWIND does not accept it. Repetition stops at the first nonzero return code. R15 uses the common physical-tape namespace: `0` success, `12` tape mark during block spacing, `16` EOI boundary, `24` invalid state, `28` tape/host I/O, `32` format, `36` internal/resource, and `40` BOT boundary. `$CONTROL` preserves CC.

B.24 ABEND

```
ABEND code
ABEND (register)
```

Optional operands include:

```
DUMP
STEP
USER
SYSTEM
REASON=value
REASON=(register)
```

`DUMP`, `STEP`, `USER`, and `SYSTEM` are positional ABEND options. They may appear in any order among themselves, but all must precede the `REASON=` keyword. This follows the global rule that positional operands precede keyword operands.

The effective ABEND code is the low 12 bits.

Defaults are:

```
no DUMP
no STEP
USER
REASON=0
```

The reason value is 31 bits.

TSDSUTIL-generated fatal runtime failures use:

```
SYSTEM=EEE
REASON=TSDS diagnostic number
```

On Linux, the ABEND host exit status defaults to 100 and can be changed with `ABEND_STATUS`.

Open DCBs are automatically closed as part of termination cleanup; a cleanup error does not replace the original ABEND.

B.25 CLOSE

```
CLOSE (dcb)
CLOSE (dcb1,dcb2,...)
CLOSE dcb
CLOSE (dcb1,(REWIND),dcb2,(FREE),...)
```

One bare DCB address or a nonempty structural list is accepted. Within a structural list, a DCB may be followed by the one-element option list `(REWIND)` or `(FREE)`; option names are case-insensitive. Entries are processed left-to-right and the list operation is not atomic. Processing stops at the first recoverable state or host-close failure, leaving prior successful closes in effect. `FREE` is retained where specified for OS-specific use.

R15 returns the CLOSE status; other registers and CC are unchanged.

CLOSE flushes pending blocked output where required and completes tape trailer/tape-mark processing for usable output data sets.

Fatal CLOSE conditions ABEND when the data set cannot be left in a usable condition. Recoverable CLOSE conditions return the defined RC.

After CLOSE, DCB attributes that were merged during OPEN are restored to their current baseline values. `$MODDCB` changes the closed-DCB baseline, so CLOSE does not necessarily restore the values originally assembled.

B.26 FREEMAIN

```
FREEMAIN R,LV=value,A=(register)
FREEMAIN RC,LV=value,A=(register)
```

`LV` can be an immediate value or register-derived value as supported by the macro.

FREEMAIN releases only a complete, currently live GETMAIN allocation. Both the starting address and exact allocation size are validated.

`R` is the unconditional form: a mismatch is fatal.

`RC` is the conditional form: an invalid release returns a nonzero R15 and diagnostic information rather than unconditionally ABENDing.

Freed virtual mappings are destroyed.

With GETMAIN debugging enabled, diagnostics include allocation address, length, and allocation history information.

B.27 GET

```
GET dcb-address,buffer-address
```

Reads one **logical record** according to the open DCB.

For F/FB, the logical record length is LRECL. For V/VB, the logical record includes the applicable record descriptor information.

R15 values are:

```
0  record read
4  EOF
8  read error
```

If an INPUT DCB has EODAD, logical EOF branches to EODAD according to the DCB processing rules. EODAD is optional; without it, EOF is returned in R15.

GET supports logical tape processing as well as ordinary host-file processing when the DCB attributes permit it. Tape blocking/deblocking is handled by TSDSUTIL.

R0–R14 and CC are unchanged.

B.28 GETMAIN

```
GETMAIN R,LV=value
GETMAIN R,LV=(register)
GETMAIN RC,LV=value
GETMAIN RC,LV=(register)
```

Allocates one whole virtual-storage block.

The allocated address is returned according to the GETMAIN interface. Newly allocated and reused storage is initialized to the configured `GETMAIN_FILL` byte, X'00' by default.

`R` is the unconditional form.

`RC` is the conditional form for requests where the program wants a return code rather than unconditional failure handling.

A runtime-register length whose signed 32-bit value is negative is an invalid length (TSDS8010). An ordinary non-relocatable expression is an unsigned assembly-time request; a request that cannot fit in the available region is reported as TSDS8011.

`#PRAGMA GETMAIN_DEBUG` adds allocation diagnostics. TRACE records successful allocations with address, length, and allocation identifier.

Execution statistics track GETMAIN calls, current allocation, peak allocation, largest allocation, and outstanding allocations.

B.29 OPEN

```
OPEN (dcb)
OPEN (dcb,INPUT)
OPEN (dcb,OUTPUT)
OPEN (dcb,EXTEND)
```

Multiple DCBs can be supplied according to the OPEN list syntax. List entries are processed left-to-right and the operation is not atomic. If an earlier OPEN succeeds and a later entry returns the recoverable TSDS8021 unallocated-DD status, the earlier DCB remains open. Fatal OPEN failures ABEND and termination cleanup closes already-open DCBs.

If the option is omitted, INPUT is assumed.

At OPEN, TSDSUTIL merges supplied DCB, DD/allocation, and applicable tape-label attributes. If the DCB has a nonzero `OPENEXIT`, the exit is invoked **before** generic OPEN defaults are supplied. PATH, DUMMY, and new/unlabeled output tape therefore present zero for any unsupplied RECFM/LRECL/BLKSIZE field. DATA allocation defaults FB/80/3280 are intrinsic and are already present, while existing labeled tape contributes its label-derived attributes.

OPENEXIT receives `R1=DCB address` and `R14=X'00FFFFFE'`; `BR R14` is the normal return and R15 is ignored. Within the exit, `$MODDCB` may change only RECFM/LRECL/BLKSIZE. After return, remaining zero fields receive normal generic defaults, final validation runs, and the persistent JFCB is synchronized before OPEN completes. `X'FFFFFE'` is reserved for this active return context and causes SYSTEM EEE if reached outside it.

OPEN, CLOSE, RDJFCB, GET, PUT, `$READ`, `$WRITE`, `$CONTROL`, and `$EOJ` are prohibited while OPENEXIT is active; so are exit-time `$MODDCB` changes to DDNAME or OPENEXIT. These protected-environment violations cause SYSTEM EEE.

The final effective values remain visible in the DCB while it is open. CLOSE restores the current baseline values, including prior closed-state `$MODDCB` changes; persistent JFCB values are not rolled back.

An unallocated or missing DDNAME, diagnostic TSDS8021, is the only recoverable OPEN failure and is returned in R15. Every other OPEN failure causes a SYSTEM=EEE ABEND with REASON equal to the TSDS diagnostic number.

CC is unchanged.

B.30 PUT

```
PUT dcb-address,buffer-address
```

Writes one **logical record** according to the DCB attributes.

For blocked formats, TSDSUTIL performs the required blocking. A final partial output block is flushed at CLOSE where applicable.

PUT supports logical tape output as well as ordinary host-file output when the DCB attributes permit it.

R15 returns the PUT status; R0–R14 and CC are unchanged.

B.31 WTO

Quoted forms:

```
WTO 'message'
WTO "message"
```

Storage form:

```
WTO text-address,text-length
```

Writes a host-service/operator-style message.

The maximum message length is 126 bytes.

Quoted text is converted according to the normal TSDSUTIL character rules. The storage form uses the supplied EBCDIC text.

Linux terminal behavior

On Linux, WTO does **not** write to standard output. It opens:

```
/dev/tty
```

for read/write access and writes the converted message directly to the process's controlling terminal, followed by a newline.

This means WTO continues to address the active terminal even when standard output has been redirected to a file or pipe.

If the process has no usable controlling terminal—for example, some batch, daemon, or detached execution environments—`/dev/tty` cannot be opened. In that case WTO reports the TSDSUTIL runtime **TTY unavailable** condition.

If `/dev/tty` can be opened but the message cannot be written, flushed, or closed successfully, WTO reports the runtime **TTY I/O** failure.

B.32 WTOR

Quoted forms:

```
WTOR 'message',reply-address,reply-length
WTOR "message",reply-address,reply-length
```

Storage form:

```
WTOR text-address,text-length,reply-address,reply-length
```

Writes a message and obtains a host reply.

The maximum message length is 126 bytes.

The maximum reply length is 119 bytes.

Linux terminal behavior

On Linux, WTOR opens:

```
/dev/tty
```

for read/write access. It writes the message to the process's controlling terminal, appends a newline, flushes the terminal, and then reads the reply from that same `/dev/tty`.

WTOR therefore does **not** read its reply from standard input. This is important because standard input is normally being used to supply the TSDSUTIL source program.

The reply line is consumed through its terminating newline. At most the caller-specified reply capacity is retained; any additional characters on that terminal line are discarded. The newline itself is not returned to the program. Retained native characters are converted to EBCDIC before being stored in the supplied reply area.

If there is no usable controlling terminal, `/dev/tty` cannot be opened and WTOR reports the runtime **TTY unavailable** condition.

If terminal output, input, flush, or close fails, WTOR reports the runtime **TTY I/O** failure.

Programs intended to run unattended should therefore avoid WTOR unless the execution environment is known to provide a controlling terminal.

B.33 Logical versus Physical I/O

Two separate interfaces are intentionally provided for tape:

```
GET / PUT        logical-record interface
$READ / $WRITE  physical-block interface
```

GET and PUT honor RECFM and perform blocking/deblocking.

`$READ` and `$WRITE` operate on physical tape blocks and require the physical MACRF forms:

```
R
W
RW
```

`RECFM=U` is reserved for the physical interface rather than GET/PUT record processing.

A tape mark written with:

```
$WRITE dcb,TYPE=TM
```

counts as a write event. A physical tape mark returned by `$READ` counts as a read event.

B.34 Macro Register and CC Conventions

Macros are designed to change only the registers documented by their interfaces.

Important examples include:

```
$LINENO    R15
$RAND      R1
$TIMEUSD   R0, R1
$READ      R1, R15
$CONTROL   R15
$SHOWDCB   R15
$TESTDCB   R15 and, on success, CC
GET        R15
OPEN       R15 for recoverable status
PUT        R15
```

Diagnostic macros such as `$REGS`, `$DUMP`, and `$DFMT` do not intentionally alter program state.

When a macro does not document a CC result, CC is preserved.

B.35 Macro List Completeness

The current executable macro table contains these 39 names:

```
$RETURN  $LINENO  $DUMP     $EOJ      $PRINT    $RAND
$REGS    $RESTORE $SAVE     $SLEEP    $SRAND    $TIMEUSD
TIME     $VALDATE $VALTIME  $CVTDATE  $DAYDIFF  $DAYADJ
$EXPDATE $FMTDATE $TIMEADJ $TRACE    $ASSERT   $DFMT     $SHOWDCB
$TESTDCB $MODDCB  $READ     $WRITE    $CONTROL  ABEND     CLOSE
GET      OPEN     PUT       GETMAIN   FREEMAIN  WTO       WTOR
```

This list is authoritative for the current release. `#PRAGMA`, `#IF`, `#COPY`, `DC`, `DS`, `DSECT`, `EQU`, `START`, and similar language elements are assembler directives or definitions rather than executable macro instructions.

The service-instruction reference also includes `DEVTYPE` and `RDJFCB`; see B.45 and B.46.

B.36 TIME DEC

```
[label] TIME DEC
```

Returns current configured local civil time in R0 as IBM `HHMMSSth` (hundredths, no sign nibble) and JDATE in R1 as `0CYYDDDF`. `TIME DEC` observes the configured timezone and DST rules and preserves CC.

B.37 $VALDATE and $VALTIME

```
$VALDATE TYPE=JDATE|GDATE,DATE=operand
$VALTIME TIME=operand
```

Validation operands accept `label`, `0(Rx)`, or `(Rx)`. R15 is 0 for valid input. Invalid JDATE/GDATE/TIME returns 4/8/12 respectively. Validation includes the encoded representation as well as calendar/time range rules.

B.38 $CVTDATE

```
$CVTDATE FROMTYPE=JDATE|GDATE|STIME|STCK|STCKE,FROM=operand,
         TOTYPE=JDATE|GDATE|STIME|STCK|STCKE,TO=operand
        [,FROMTIME=operand][,TOTIME=operand]
```

JDATE/GDATE may use `label`, `0(Rx)`, or `(Rx)`; STIME/STCK/STCKE are storage-only. `FROMTIME=` is an optional civil-time input for JDATE/GDATE to a timestamp and defaults to midnight. `TOTIME=` is an optional civil-time output for timestamp to JDATE/GDATE. Direct STIME/STCK/STCKE conversions are pure UTC representation changes. `FROMTYPE` and `TOTYPE` must be different; a same-type conversion is rejected at assembly time.

B.39 $DAYDIFF

```
$DAYDIFF jdate1,jdate2
```

Returns signed 32-bit `jdate1-jdate2` calendar days in R1 and status in R15. Timezone, DST, and time-of-day do not participate.

B.40 $DAYADJ

```
$DAYADJ jdate,signed-days,result
```

Adds the signed 32-bit day value to JDATE and stores a new JDATE. Negative values subtract. All operands support `label`, `0(Rx)`, or `(Rx)`, except direct output `(R15)` is invalid. Range failure returns RC 20 and leaves the result unchanged.

B.41 $TIMEADJ

```
$TIMEADJ input-stime,result-stime[,SECONDS=s][,MINUTES=m][,HOURS=h][,DAYS=d]
```

At least one adjustment keyword is required. STIME operands are storage-only. `SECONDS=`, `MINUTES=`, `HOURS=`, and `DAYS=` are signed assembly-time expressions evaluated when the macro is encoded; they are not runtime register/storage operands. The adjustment is `s + m*60 + h*3600 + d*86400`; timezone and DST are not consulted. Range underflow/overflow returns RC 20 without changing the result.

B.42 $EXPDATE

```
$EXPDATE jdate,exp-area
```

Expands a JDATE into the built-in 37-byte EBCDIC `$_EXPDATE` mapping. Use:

```
#COPY $_EXPDATE
```

to define the DSECT. It contains numeric year/month/day/JDAY, full and short month and weekday names, Sunday-based EBCDIC weekday `0-6`, and EBCDIC leap indicator `Y/N`.

B.43 $FMTDATE

```
$FMTDATE jdate[,time],FORMAT='literal'|"literal",AREA=address,AREALEN=length
$FMTDATE jdate[,time],FORMAT=address,FMTLEN=length,
         AREA=address,AREALEN=length
```

An inline FORMAT literal may use matching single or double quotes; doubled matching quote characters escape that delimiter. TIME defaults to midnight. Supported conversions are `%Y %y %m %d %j %A %a %B %b %H %I %M %S %p %%` plus TSDSUTIL `%h` for hundredths. Successful output is EBCDIC and blank-fills the complete output field. RC 36 means the complete result does not fit; RC 40 means invalid format. Either error leaves the output area unchanged.

B.44 Date/Time Register and Error Conventions

The shared date/time RC meanings are 0 success, 4 invalid JDATE, 8 invalid GDATE, 12 invalid TIME, 16 invalid STIME/STCK/STCKE value, 20 range, 24 invalid runtime operand/parameter, 28 nonexistent DST local time, 32 ambiguous DST local time, 36 output area too small, and 40 invalid format.

Date/time macros preserve CC. Inputs and effective addresses are captured before outputs are committed. On a date/time RC failure only R15 changes. Direct register output `(R15)` is invalid; indirect `0(R15)` uses R15's entry value.

B.45 DEVTYPE

Syntax

```
DEVTYPE ddname,return-area
```

Both positional operands are required. `ddname` addresses an 8-byte blank-padded DDNAME and `return-area` addresses an 8-byte writable area. Ordinary storage addresses and structural `(Rx)` runtime-address forms are accepted.

DEVTYPE clears all eight return bytes before lookup. R15=0 means the DD is defined; R15=4 means the DD is undefined. Invalid operand storage is an addressing failure.

Resource	Return bytes
3420-style AWS tape	3210800300007FF8
DATA / simulated SYSIN	0000010200007FF8
Linux/PATH	0000010300007FF8
defined DUMMY	0000000000000000

B.46 RDJFCB

Syntax

```
RDJFCB dcb,jfcb-area
```

Both positional operands are required. The DCB operand and 176-byte writable return area use the normal address rules, including structural `(Rx)` runtime addresses. The return area is zero-initialized before semantic lookup.

R15=0 returns the current allocation/JFCB image. R15=4 means the DCB itself is valid but its DDNAME is blank, unusable, or not currently allocated. Invalid DCB or result-area storage is an addressing failure.

Supported fields are `JFCBDSNM` (+0, 44), `JFCBLTYP` (+42), `JFCBFLSQ` (+44), `JFCDSORG` (+62), `JFCRECFM` (+64), `JFCBLKSI` (+66), `JFCLRECL` (+68), `JFCBNVOL` (+75), and `JFCBVOLS` (+76). Unsupported bytes remain zero. `#COPY $_JFCB` supplies the built-in 176-byte DSECT with `JFCBLEN=176`.

PATH uses the host path, right-end truncated as `...` plus the final 41 characters when necessary. DATA uses `SYSIN.SI001` through `SYSIN.SI999`; DUMMY uses `NULLFILE`. Tape returns the applicable data-set, label, sequence, volume, DSORG, and record information.

Before OPEN the image reflects allocation state. A successful OPEN synchronizes final effective record attributes and applicable tape sequence into persistent JFCB state. CLOSE does not roll that state back. RDJFCB is prohibited during an active OPENEXIT.

Appendix C — Data Types, Constants, and Storage Definitions

Applies to: TSDSUTIL Version 1.0.1

This appendix is the compact reference for `DC`, `DS`, constant types, storage lengths, alignment, repeat factors, address constants, literals, and related attributes.

The forms documented here have been checked against the current data-definition parser.

C.1 DC and DS

`DC` allocates storage and supplies an initial value:

```
COUNT    DC    F'100'
NAME     DC    CL10'MICKEY'
```

`DS` allocates storage without supplying an application initializer:

```
COUNT    DS    F
NAME     DS    CL10
```

All ordinary data definitions must appear before the first `START`. Once executable code has begun, ordinary `DC`, `DS`, DSECT, and related data definitions cannot resume. Assembler-generated literals are the exception.

A `DS 0type` statement can be used as an aligned zero-length label, including in CODE where that form is permitted:

```
EOF      DS    0H
```

C.2 Type Summary

Type	Meaning	Natural length	Alignment	Explicit length
`C`	EBCDIC character	initializer length	1	`CLn`
`X`	Hexadecimal bytes	digits rounded to bytes	1	`XLn`
`B`	Binary bit string	bits rounded to bytes	1	`BLn`
`P`	Packed decimal	minimum required	1	`PLn`
`H`	Fixed-point halfword	2	2	none
`F`	Fixed-point fullword	4	4	none
`D`	8-byte fixed integer	8	8	none
`FD`	8-byte fixed integer form	8	8	none
`A`	Address constant	4	4	`AL1`–`AL4` are separate forms
`AL1`	Explicit-length address	1	1	fixed
`AL2`	Explicit-length address	2	1	fixed
`AL3`	Explicit-length address	3	1	fixed
`AL4`	Explicit-length address	4	1	fixed
`V`	VCON/linkage address constant	4	4	DC only

`V` is a linkage-time constant used with translation-unit ENTRY/VCON processing. It is not a `DS` storage type.

C.3 Repeat Factors

A decimal repeat factor precedes the type:

```
ARRAY    DS    10F
FILL     DC    3C'Z'
PACKED   DC    3P'7'
```

The repeat applies to the complete element.

Examples:

```
DS 10F       ten 4-byte fullwords
DC 3C'Z'     three 1-byte EBCDIC Z values
DC 3P'7'     three packed-decimal elements
```

A zero repeat allocates no element bytes.

Repeat factors are decimal integers.

C.4 Multiple DC Items

One `DC` statement can define several items separated by commas:

```
VALUES   DC    H'2',F'3',H'4'
```

Each item receives its own natural alignment. In this example, padding can be inserted before the fullword and before later items as required by their types.

The label on the statement identifies the first generated item.

This feature makes it possible to build a structured area from several different constant types on one statement, although separate labeled statements are often clearer when individual fields need names.

C.5 Character Constants — C and CLn

Natural-length character constant:

```
TEXT     DC    C'HELLO'
```

Explicit-length character constant:

```
NAME     DC    CL10'MICKEY'
```

Character constants are converted from the source character set to **EBCDIC CP037** when assembled.

A `C` constant without an explicit length occupies the number of resulting characters.

`CLn` occupies exactly `n` bytes.

If the initializer is shorter than the declared length, the remaining bytes are padded with EBCDIC blanks (`X'40'`):

```
FIELD    DC    CL5'XY'
```

produces the EBCDIC equivalent of:

```
"XY   "
```

If the initializer is longer than the declared `CLn`, assembly reports an error; it is not silently truncated.

A quote character inside a quoted character initializer is represented by doubling that same quote according to the accepted source syntax.

The source-to-EBCDIC conversion accepts only characters representable by the CP037 mapping. An unsupported source character is an assembly error rather than being silently substituted.

C.6 Character Storage — DS C and CLn

`DS C` reserves one byte:

```
FLAG     DS    C
```

`DS CLn` reserves exactly `n` character bytes:

```
BUFFER   DS    CL80
```

Character storage is byte aligned.

The reserved bytes are part of TSDSUTIL DATA storage but `DS` does not assert an application character value for them.

C.7 Hexadecimal Constants — X and XLn

Hexadecimal constants contain hexadecimal digits:

```
BYTE     DC    X'FF'
WORD     DC    X'1234'
```

An odd number of hexadecimal digits is allowed and is right aligned within the natural byte representation.

For example:

```
NIBBLE   DC    X'4'
```

produces:

```
X'04'
```

An explicit length can be supplied:

```
VALUE    DC    XL4'F'
```

which produces:

```
X'0000000F'
```

Hexadecimal values are right aligned and high-order bytes are zero filled.

If the natural hexadecimal value requires more bytes than an explicit `XLn` provides, assembly reports an error.

Hexadecimal storage is byte aligned.

C.8 Hexadecimal Storage — DS X and XLn

`DS X` reserves one byte:

```
BYTE     DS    X
```

`DS XLn` reserves the requested byte length:

```
WORK     DS    XL36
```

This form is useful for arbitrary binary areas, save/work areas, returned macro vectors, and physical tape buffers.

C.9 Binary Constants — B and BLn

Binary constants contain only `0` and `1`:

```
BITS     DC    B'101'
```

The bit string is right aligned in the natural byte field. Thus:

```
BITS     DC    B'101'
```

produces:

```
X'05'
```

An explicit byte length can be supplied:

```
BITS     DC    BL2'1'
```

which produces:

```
X'0001'
```

If the bit string requires more bytes than the declared `BLn`, assembly reports an error.

Binary data is byte aligned.

C.10 Binary Storage — DS B and BLn

`DS B` reserves one byte.

`DS BLn` reserves exactly `n` bytes:

```
MASKS    DS    BL8
```

The `L` length is measured in **bytes**, even though the initializer syntax for `DC B` is expressed as bits.

C.11 Packed Decimal — P and PLn

Packed decimal constants contain decimal digits with an optional leading sign:

```
ONE      DC    P'1'
NEG      DC    P'-45'
DATE     DC    PL5'202608'
```

The sign occupies the low-order nibble of the final byte.

Positive values use a positive packed sign; negative values use the negative packed sign.

Examples from the encoding rules:

```
P'1'       -> X'1C'
PL2'1'     -> X'001C'
P'-45'     -> X'045D'
```

Without an explicit length, `P` uses the minimum byte length needed for the digits plus sign.

`PLn` occupies exactly `n` bytes.

Digits are right aligned. If a `PLn` field is too short for all high-order digits, the high-order excess digits are intentionally truncated.

For example:

```
SMALL    DC    PL1'123'
```

retains only the digits that fit with the sign and produces:

```
X'3C'
```

This truncation behavior is intentional and differs from the error behavior of overlength C, X, and B explicit-length constants.

Packed decimal is byte aligned.

C.12 Packed Storage — DS P and PLn

`DS P` reserves one byte.

`DS PLn` reserves exactly `n` packed-decimal bytes:

```
AMOUNT   DS    PL6
```

A `DS` field is only storage; its bytes are not guaranteed to contain a valid packed-decimal value until the program places one there.

Diagnostic facilities such as `$DFMT` therefore validate packed data when interpreting it and report invalid packed fields explicitly.

C.13 Halfword Fixed Integer — H

`H` is a 2-byte fixed integer aligned on a 2-byte boundary:

```
COUNT    DC    H'100'
WORK     DS    H
```

Numeric fixed-point constants are signed by default.

Example:

```
MINUS1   DC    H'-1'
```

produces:

```
X'FFFF'
```

The signed value must fit the 16-bit signed range unless unsigned interpretation is explicitly requested.

`H` does not accept a length modifier.

C.14 Fullword Fixed Integer — F

`F` is a 4-byte fixed integer aligned on a 4-byte boundary:

```
COUNT    DC    F'100'
WORK     DS    F
```

The value is signed by default and stored in big-endian System/370 byte order.

The signed value must fit the 32-bit signed range unless unsigned interpretation is explicitly requested.

`F` does not accept a length modifier.

Fullword constants are commonly used for counters, lengths, return values, and ordinary binary arithmetic.

C.15 Doubleword Integer Forms — D and FD

`D` and `FD` each occupy 8 bytes and are aligned on an 8-byte boundary:

```
VALUE1   DC    D'17'
VALUE2   DC    FD'18'
WORK1    DS    D
WORK2    DS    FD
```

These are fixed 8-byte integer representations for the data-definition facility.

The type distinction is retained in metadata and facilities such as DSECT formatting even though both occupy eight bytes.

The signed value must fit the supported 64-bit signed range unless unsigned interpretation is explicitly requested.

Neither form accepts a length modifier.

C.16 Unsigned H/F/D/FD Constants

H, F, D, and FD constants are signed by default.

Explicit unsigned interpretation is supported for a numeric initializer. One accepted form places `U` at the beginning of the quoted numeric value:

```
MAXH     DC    H'U65535'
```

which permits the full unsigned 16-bit range and produces:

```
X'FFFF'
```

Unsigned range checking is based on the width of the type.

The same principle applies to the supported F, D, and FD fixed-width numeric forms.

Use unsigned form when the intended bit pattern is a nonnegative binary value that exceeds the signed range of the field.

C.17 Numeric Initializers

Quoted H/F/D/FD initializers can identify:

- A decimal integer.
- An `X'...'` numeric expression spelling.
- A `B'...'` numeric expression spelling.
- An EQU symbol.

Examples:

```
A        DC    F'100'
B        DC    F'-1'
C        DC    F'MAXIMUM'
```

The resolved numeric initializer must be non-relocatable.

General arithmetic expressions inside the quoted numeric initializer are intentionally not accepted. Use an EQU symbol when a computed assembly-time value is needed:

```
VALUE    EQU   BASEVALUE+4
FIELD    DC    F'VALUE'
```

Forward EQU resolution is permitted where the normal deferred-expression rules allow it.

C.18 Address Constant — A

`A` defines one or more 4-byte address values:

```
PTR      DC    A(BUFFER)
```

A list is allowed:

```
TABLE    DC    A(ONE,TWO,THREE)
```

Each A element is four bytes and receives normal 4-byte alignment.

The expression is evaluated as an address/value expression under the assembler relocation rules.

A forward reference can be used when it can be resolved during the later assembly pass:

```
PTR      DC    A(FUTURE)
...
FUTURE   EQU   4096
```

TSDSUTIL program addressing is 24-bit, but the A field itself is four bytes.

C.19 Explicit-Length Address Constants — AL1 through AL4

Explicit-length address constants store the low-order requested number of bytes:

```
A1       DC    AL1(1,2,255)
A2       DC    AL2(VALUE)
A3       DC    AL3(BUFFER)
A4       DC    AL4(BUFFER)
```

The permitted lengths are exactly:

```
1 2 3 4
```

AL values are byte aligned; they do not receive A's natural 4-byte alignment.

Negative or larger values are represented in the selected low-order byte width according to the address-constant encoding rules.

The explicit type is retained. `$DFMT`, for example, can distinguish `A` from `AL3`.

C.20 VCON — V

`V` defines a four-byte linkage address constant:

```
VPTR     DC    V(SERVICE)
```

The initializer requires parentheses.

A VCON is resolved during final translation-unit linkage against an appropriate ENTRY rather than as an ordinary same-translation-unit address constant.

`V` is aligned like a four-byte address constant.

This form is available on `DC`; it is not a `DS` storage type.

Translation units, ENTRY, and VCON linkage are described in Chapter 8.

C.21 Alignment Summary

Natural data alignment is:

```
byte-oriented C/X/B/P/AL     1 byte
H                            2 bytes
F                            4 bytes
A                            4 bytes
D                            8 bytes
FD                           8 bytes
V                            4 bytes
```

Alignment can insert unused bytes between consecutive definitions.

Example:

```
A        DC    H'2'
         DC    F'3'
         DC    H'4'
```

The F value aligns to the next 4-byte boundary, so padding is inserted after the first H.

Alignment affects layout and symbol offsets but TSDSUTIL does not impose a blanket natural-alignment requirement on runtime machine-instruction references.

C.22 DS 0type

A repeat factor of zero can establish alignment without allocating bytes:

```
AREA     DS    0F
```

The location counter advances only as necessary to satisfy the type's alignment.

Common forms include:

```
         DS    0H
         DS    0F
         DS    0D
```

This is frequently used to establish aligned labels.

After `START`, `DS 0type` is the special permitted form for defining an aligned CODE label; it does not create DATA storage.

C.23 DSECT Field Definitions

DSECT fields use the same DS type and alignment rules:

```
RECORD      DSECT ,
NAME        DS    CL20
COUNT       DS    F
AMOUNT      DS    PL5
```

The DSECT describes offsets; it does not allocate an instance of the record in DATA storage.

The named field metadata—including type, offset, and length—is available to facilities such as `$DFMT`.

Unnamed fields can be used to create spacing or layout but have no symbol that the program can reference.

EQU definitions in a DSECT have length zero and do not consume field bytes.

C.24 Character Quotes

Both supported quote styles can be used where the data-definition parser permits a quoted initializer.

A quote matching the current delimiter can be represented by the accepted doubled-quote convention.

Character content is converted to EBCDIC CP037 at assembly time.

The quote characters themselves are source delimiters and are not stored unless represented as data within the initializer.

C.25 Length Attribute — L'

TSDSUTIL supports the assembler-style length attribute:

```
L'symbol
```

For example:

```
BUFFER   DS    CL80
...
         $DUMP BUFFER,L'BUFFER
```

`L'BUFFER` evaluates to 80.

For a symbol defined by a normal DC/DS item, the length is that symbol's defined element length.

For an EQU symbol, the length is zero.

The length attribute is especially useful for buffers and macro operands because changing the definition automatically changes the value used by later references.

C.26 Labels on Multi-Element Definitions

A label on a definition identifies the location and element metadata associated with the first item represented by that label.

For example:

```
VALUES   DC    H'2',F'3',H'4'
```

`VALUES` identifies the first H element, not the total storage span of every comma-separated item.

When the total structure length is important, use explicit labels or an end label so the intended extent is clear.

C.27 Literals

A literal begins with `=` and requests an assembler-managed constant:

```
         CLC   FIELD,=C'ABC'
         L     R3,=F'100'
```

Literals use the supported data-constant syntax and are allocated by the assembler.

They are the one exception to the rule that ordinary data definitions cannot be added after `START`: the programmer can reference a new literal in executable code because the assembler manages literal allocation automatically.

Literal identity includes its type spelling. For example, constants that happen to produce the same bytes are not automatically the same literal when their source type/length definitions differ.

Literal text is data, not a source symbol reference merely because it contains symbol-like characters.

C.28 DC versus Literal

Use `DC` when the program needs a named data object:

```
LIMIT    DC    F'100'
```

Use a literal when a constant is needed only as an operand:

```
         C     R3,=F'100'
```

A named DC is easier to share, modify, display, and cross-reference. A literal is convenient for an otherwise anonymous constant.

Both occupy controlled TSDSUTIL virtual storage rather than host C-language memory.

C.29 Endianness

Numeric binary and address values are stored in System/370-style big-endian byte order.

For example:

```
VALUE    DC    F'16'
```

is represented as:

```
X'00000010'
```

and:

```
PTR      DC    A(4096)
```

is represented as:

```
X'00001000'
```

This allows System/370 machine instructions to operate on the bytes using the expected representation.

C.30 Type and Length Errors

Common assembly-time errors include:

- `CL0`, `XL0`, `BL0`, or `PL0`.
- An AL length other than 1, 2, 3, or 4.
- Applying an unsupported length modifier to H/F/D/FD.
- Character initializer longer than `CLn`.
- Hexadecimal value requiring more bytes than `XLn`.
- Binary value requiring more bytes than `BLn`.
- Invalid hexadecimal, binary, or packed digits.
- Signed/unsigned H/F/D/FD value outside the representable range.
- Relocatable value used where a fixed numeric initializer is required.
- Invalid or unresolved address/linkage expression.

Packed `PLn` is a deliberate exception to the usual overlength rule: high-order excess decimal digits are truncated to fit the explicit packed field.

C.31 Compact Examples

```
CHAR1    DC    C'ABC'
CHAR2    DC    CL5'XY'

HEX1     DC    X'4'
HEX2     DC    XL4'F'

BIN1     DC    B'101'
BIN2     DC    BL2'1'

HALF     DC    H'-1'
UHALF    DC    H'U65535'
FULL     DC    F'16'
DBL1     DC    D'17'
DBL2     DC    FD'18'

PACK1    DC    P'1'
PACK2    DC    PL2'1'
PACK3    DC    PL1'123'

PTR      DC    A(BUFFER)
SHORT    DC    AL2(-1)
LINKPTR  DC    V(SERVICE)

BUFFER   DS    CL80
WORK     DS    XL36
TABLE    DS    10F
```

These examples illustrate the principal current data-definition forms. Chapter 3 explains their use in program layout; this appendix serves as the compact syntax and representation reference.

C.32 Date/Time Representations

JDATE is a four-byte packed value `0CYYDDDF`. The high-order nibble is zero, `C` is a decimal century digit 0-9, `YY` is the year within that century, `DDD` is day-of-year 001-365/366, and the low-order `F` is the positive sign nibble. The civil year is `1900 + C*100 + YY`, so the supported range is 1900-2899.

GDATE is four bytes of unsigned packed decimal `YYYYMMDD`, valid from 19000101 through 28991231.

TIME is four bytes of unsigned packed decimal `HHMMSSth`: hour 00-23, minute/second 00-59, tenths `t`, hundredths `h`. Leap second 60 is not supported.

STIME is an eight-byte unsigned binary count of whole seconds elapsed since 1900-01-01 00:00:00 UTC. It has no timezone or DST state. Arithmetic on STIME is elapsed-time arithmetic. STIME is storage-only in date/time macros (`label` or `0(Rx)`).

STCK is the classic eight-byte IBM Time-of-Day clock representation, based on UTC and the 1900 epoch. Its classic 64-bit epoch rolls over at 2042-09-17 23:53:47.370496 UTC; TSDSUTIL does not infer a later epoch.

STCKE is the sixteen-byte extended TOD representation used when the classic STCK epoch is insufficient.

The configured timezone/DST rules affect only conversion between absolute UTC timestamp values (STIME/STCK/STCKE) and local JDATE/GDATE/TIME values. Direct conversions among the timestamp types never apply timezone or DST.

C.33 Expression-Based Explicit Lengths

For the variable-length character, hexadecimal, binary, and packed-decimal forms, an explicit length can be written either as the traditional decimal suffix or as a parenthesized assembly expression. This applies to both `DS` and `DC`:

```
CL40              CL(expression)
XL16              XL(expression)
BL8               BL(expression)
PL6               PL(expression)
```

Examples:

```
ZERO      DS    0C
DATEINFO  DS    CL40
FLEN      EQU   *-ZERO
AREA2     DS    CL(FLEN)
AREA3     DS    XL(FLEN+8)
BUFFER    DC    CL(FLEN)' '
AREA4     DS    BL(L'DATEINFO)
```

A length expression affects the location counter immediately. It therefore must be completely resolvable at the statement where it is used, must be absolute (non-relocatable), and must evaluate to a positive value representable as a TSDSUTIL storage length. Forward references are not deferred for an explicit DS/DC length. Invalid, unresolved, relocatable, zero, or negative length expressions produce `TSDS5003E`.

Appendix D — Return Codes and ABENDs

Applies to: TSDSUTIL Version 1.0.1

This appendix summarizes the user-visible return codes, completion codes, and ABEND conventions used by TSDSUTIL.

A return code in R15 belongs to the macro or service that just executed. It is not automatically the program's final completion code. A program can test the service result, continue processing, and later select a different `$EOJ` value.

Fatal runtime failures use a SYSTEM ABEND rather than a recoverable service RC when the interface defines the condition as nonrecoverable.

D.1 Program Completion Codes

Normal execution ends with `$EOJ`.

Explicit completion code:

```
$EOJ  8
```

Register completion code:

```
$EOJ  (R5)
```

Current-R15 completion code:

```
$EOJ  ,
```

`$EOJ ,` and `$EOJ (R15)` are equivalent.

On Linux, the logical `$EOJ` completion code is returned through the low-order 8 bits of the host process exit status.

D.2 Linux Driver Status Values

When no normal program completion code or ABEND host status applies, the Linux driver uses these general statuses:

Host status	Meaning
0	Successful TSDSUTIL processing
8	TSDSUTIL error
16	Severe TSDSUTIL/driver error

Examples of status 16 conditions include malformed command-line usage and severe failures that prevent normal assembly/execution processing.

An ABEND uses the configured ABEND host status instead of the ordinary 8/16 processing status.

D.3 ABEND_STATUS

The Linux host status used for an ABEND defaults to:

```
100
```

It can be changed with:

```
#PRAGMA ABEND_STATUS n
```

where `n` is 0 through 255.

This host process status is separate from:

- The USER or SYSTEM ABEND classification.
- The 12-bit ABEND code.
- The 31-bit ABEND reason.

D.4 USER ABEND

A program can request an abnormal termination:

```
ABEND 1234
```

USER is the default ABEND type.

The effective ABEND code is the low-order 12 bits of the supplied code.

The default reason is zero.

Optional ABEND operands can request the supported DUMP/STEP flags, USER/SYSTEM classification, and an explicit reason. The standalone ABEND options may appear before or after `REASON=`.

For example:

```
ABEND 1234,USER,REASON=99
```

The exact macro syntax is summarized in Appendix B.

Beginning with V70, USER ABEND codes are displayed in **decimal**, following IBM convention. For example:

```
ABEND 1234
```

reports:

```
USER=1234
```

SYSTEM ABEND codes remain hexadecimal.

D.5 SYSTEM ABEND

TSDSUTIL uses:

```
SYSTEM=EEE
```

for runtime failures generated by the execution environment.

The REASON is normally the TSDS diagnostic number that identifies the failure.

Conceptually:

```
SYSTEM=EEE REASON=8024
```

means that TSDSUTIL terminated abnormally because runtime diagnostic 8024 was fatal.

This makes the diagnostic number the primary key for determining the cause of a TSDSUTIL-generated ABEND.

D.6 OPEN Return and ABEND Policy

OPEN uses a deliberately strict failure policy.

The only recoverable OPEN failure is:

Condition	R15 / action
DDNAME not allocated — TSDS8021	R15=8021; execution continues

Every other OPEN failure causes:

```
SYSTEM=EEE
REASON=<associated TSDS diagnostic>
```

Important examples are:

Diagnostic	Condition	Action
8021	DDNAME not allocated	Return R15=8021
8022	Host file/tape OPEN failure	ABEND

Diagnostic	Condition	Action
8023	Invalid DCB state, already open, incompatible MACRF/direction	ABEND
8024	Invalid effective DCB attributes	ABEND

This policy applies to both ordinary Linux files and AWS tape data sets.

For example, an FB DCB that resolves to:

```
LRECL=132
BLKSIZE=3990
```

ABENDs during OPEN because BLKSIZE is not an exact multiple of LRECL.

D.7 CLOSE Return and ABEND Policy

CLOSE follows a different policy from OPEN.

A CLOSE against a DCB that is not open returns the DCB-state diagnostic in R15:

```
R15=8023
```

For a Linux host file, a host `fclose()` failure returns:

```
R15=8027
```

A successful CLOSE returns:

```
R15=0
```

Tape CLOSE is stricter because failure can leave the data set structurally unusable. A tape CLOSE failure—such as inability to flush pending output or complete required tape label/tape-mark processing—causes:

```
SYSTEM=EEE
REASON=8027
```

The guiding rule is whether the data set can be left in a usable condition.

D.8 Automatic CLOSE at Normal EOJ

If `$EOJ` is reached with one or more DCBs still open, TSDSUTIL automatically closes them.

If automatic CLOSE or final tape unmount fails during otherwise normal end-of-job processing, TSDSUTIL converts the termination to:

```
SYSTEM=EEE
REASON=8027
```

The automatic-close message identifies the DCB symbol and DDNAME.

Programs should still explicitly CLOSE DCBs during normal processing.

D.9 Automatic CLOSE During ABEND

When an ABEND is already active, TSDSUTIL also attempts to close all open DCBs.

If cleanup CLOSE or tape unmount itself fails, the cleanup error is reported diagnostically but **does not replace the original ABEND code or reason**.

Thus the first abnormal condition remains authoritative.

D.10 GET Return Codes

GET performs logical-record input.

The normal R15 values are:

R15	Meaning
0	Logical record successfully read
4	Logical EOF when no EODAD routine is supplied

If EODAD is present, logical EOF transfers control to EODAD instead of returning through the next instruction with R15=4.

Runtime failures such as these are fatal rather than ordinary GET RCs:

- DCB not open for INPUT.
- Wrong MACRF.
- Invalid target storage.
- Linux text record longer than effective LRECL.
- Invalid tape logical-record/block format.
- Host/tape I/O failure.

They generate the applicable SYSTEM=EEE ABEND.

Older shorthand documentation may describe a general GET read-error value of 8; in V67 the implemented runtime error paths above are diagnosed and ABEND rather than returning a generic 8 for those failures.

D.11 PUT Return Codes

PUT performs logical-record output.

Successful PUT returns:

`R15=0`

PUT errors are generally fatal. Examples include:

- DCB not open for OUTPUT/EXTEND.
- Wrong MACRF.
- `RECFM=U` used with PUT.
- Invalid source storage.
- Invalid V/VB RDW.
- Host write failure.
- Tape logical-record/blocking failure.

These conditions generate a SYSTEM=EEE ABEND with the applicable TSDS diagnostic reason.

D.12 $SHOWDCB Return Codes

`$SHOWDCB` returns:

R15	Meaning
0	Success
4	Return area too small
8	Invalid DCB, invalid return area, or other query error

On success, the selected values are stored in the caller's return area in the order requested.

CC is not used as the `$SHOWDCB` result.

D.13 $TESTDCB Return Codes and CC

`$TESTDCB` returns:

R15	Meaning
0	Test was performed
8	Test could not be performed

When R15=0, CC contains the comparison result:

CC	Meaning
0	Equal; requested value or state is true
1	Not equal; requested value or state is false

This applies to RECFM, LRECL, BLKSIZE, DDNAME, and `OPEN=YES|NO`.

If R15 is nonzero, the previous CC is preserved.

D.14 $MODDCB Return Codes

`$MODDCB` returns:

R15	Meaning
0	All requested DCB modifications were applied
4	DCB is valid but is currently open
8	One or more requested field values are invalid
12	Request cannot otherwise be processed, such as invalid DCB

The operation is atomic. R15=4, 8, or 12 leaves the DCB unchanged.

During an active OPENEXIT, `$MODDCB` may change only provisional effective RECFM/LRECL/BLKSIZE. Attempting to change DDNAME or OPENEXIT in that protected context causes SYSTEM=EEE rather than returning a `$MODDCB` status. Outside OPENEXIT, `OPENEXIT=address|0` is a supported baseline modification.

`$MODDCB` preserves CC.

OPENEXIT protected-environment failures

While OPENEXIT is active, OPEN, CLOSE, RDJFCB, GET, PUT, `$READ`, `$WRITE`, `$CONTROL`, `$EOJ`, `$MODDCB DDNAME=...`, and `$MODDCB OPENEXIT=...` cause SYSTEM=EEE. Branching/executing at the reserved `X'FFFFFE'` sentinel outside an active OPENEXIT also causes SYSTEM=EEE. A deliberate application ABEND inside the exit retains its ordinary requested ABEND semantics rather than being converted to EEE. Invalid nonzero OPENEXIT targets use the normal invalid-executable-address failure when OPEN attempts invocation.

D.15 DEVTYPE and RDJFCB Return Codes

`DEVTYPE` returns R15=0 for a defined DD and R15=4 for an undefined DD. Its 8-byte return area is cleared before lookup. A defined DUMMY DD is successful (R15=0) even though its returned device bytes are all zero.

`RDJFCB` returns R15=0 when the 176-byte JFCB-compatible image is returned and R15=4 when the DCB is valid but has no usable/currently allocated DDNAME. The complete caller area is zero-initialized before semantic lookup.

Invalid operand storage for either service is an addressing failure rather than one of these service return codes. RDJFCB invoked during OPENEXIT is a protected-environment violation and causes SYSTEM=EEE.

D.16 Physical Tape $READ Return Codes

`$READ` uses the physical tape status namespace:

An incompatible DCB MACRF returns RC=24 and also emits a diagnostic naming the actual MACRF and the required mode (`R` or `RW` for `$READ`; `W` or `RW` for `$WRITE`; `R`, `W`, or `RW` for `$CONTROL`).

R15	Meaning
0	Data block successfully read
4	Logical EOF for SL/NL
8	Physical block longer than supplied buffer; prefix copied, residual discarded

R15	Meaning
12	Physical tape mark in BLP mode
16	Physical end-of-image/EOT
20	Read attempted after remembered EOT
24	Invalid operation or state
28	Tape/host I/O error
32	Volume/data-set format error
36	Internal/resource error

R1 receives the **full physical block length** for a data-block read.

For RC=8, R1 is the original full block length even though only the caller's buffer capacity was copied.

A maximum read length of zero is a valid consume/discard size probe. For nonempty data it returns RC=8 and places the required physical length in R1.

Invalid or protected target storage is not converted to a tape RC; it causes the normal SYSTEM address ABEND.

D.17 Physical Tape $WRITE Return Codes

`$WRITE` uses the same common tape status namespace, but only the statuses meaningful to output can occur:

R15	Meaning
0	Physical block or tape mark successfully written
8	Data block larger than effective BLKSIZE
20	Operation after EOT, where applicable
24	Invalid operation or state
28	Tape/host I/O error
32	Volume/data-set format error
36	Internal/resource error

A data length of zero returns RC=24.

`$WRITE dcb,TYPE=TM` uses the same return-code classes.

Invalid or protected source storage causes a SYSTEM address ABEND rather than a tape RC.

Physical Tape `$CONTROL` Return Codes

`$CONTROL dcb,TYPE=BSB|BSF|FSB|FSF|REWIND` uses the same common physical tape namespace as `$READ` and `$WRITE`:

R15	Meaning
0	Requested positioning completed
12	Tape mark encountered during BSB/FSB block spacing
16	Forward positioning reached physical end-of-image
24	Invalid operation or state
28	Tape/host positioning I/O error
32	Volume/data-set format error
36	Internal/resource error
40	Backward positioning reached beginning-of-tape

RC=40 and RC=16 are normal recoverable boundary conditions. `BSF/FSF` return RC=0 when the requested tape-file boundary is found. `REWIND` returns RC=0 unless the underlying positioning operation fails.

D.18 GETMAIN R Form

The unconditional form:

```
GETMAIN R,...
```

either succeeds or ABENDs.

On success, the allocated virtual address is returned in R1.

Failures generate:

```
SYSTEM=EEE
REASON=<GETMAIN diagnostic>
```

Relevant runtime diagnostics include:

Diagnostic	Meaning
8010	Invalid GETMAIN length
8011	Insufficient TSDSUTIL virtual space
8012	Host allocation failure
8034	Allocation metadata could not be recorded

D.19 GETMAIN RC Form

The conditional form:

```
GETMAIN RC,...
```

returns status instead of unconditionally ABENDing for the GETMAIN request failures.

On success:

```
R1  = allocated virtual address
R15 = 0
```

On failure:

```
R15 = applicable TSDS diagnostic number
```

Possible returned diagnostic numbers include 8010, 8011, 8012, and 8034.

A diagnostic is also issued for the conditional failure.

D.20 FREEMAIN R Form

The unconditional form:

```
FREEMAIN R,...
```

must identify the exact starting address and exact length of a currently live GETMAIN allocation.

Failure causes:

```
SYSTEM=EEE
REASON=<FREEMAIN diagnostic>
```

Relevant diagnostics include:

Diagnostic	Meaning
8013	Address is not a live GETMAIN allocation
8014	Length does not match the allocation
8032	Address points inside, rather than at the start of, a live allocation
8033	Allocation has already been freed

D.21 FREEMAIN RC Form

The conditional form:

```
FREEMAIN RC,...
```

returns status rather than unconditionally ABENDing for allocation-validation failures.

On success:

```
R15=0
```

On failure:

```
R15=<applicable FREEMAIN diagnostic number>
```

and an informational diagnostic describes the problem.

D.22 $ASSERT ABENDs

A failed assertion produces:

```
SYSTEM=EEE
REASON=8030
```

Diagnostic 8030 means the asserted comparison was false.

If the assertion cannot read the required storage, the failure is:

```
SYSTEM=EEE
REASON=8031
```

A successful `$ASSERT` does not change program-visible registers or CC.

D.23 Save-Stack ABENDs

The internal `$SAVE`/`$RESTORE`/`$RETURN` stack has fatal underflow and overflow checks.

Relevant diagnostics are:

Diagnostic	Meaning
8006	Save-stack underflow
8007	Save-stack overflow

These failures generate SYSTEM=EEE ABENDs.

D.24 Instruction-Limit ABEND

If execution reaches the configured `MAX_INSTRUCTIONS` limit:

```
TSDS8028E
```

is reported and execution terminates:

```
SYSTEM=EEE
REASON=8028
```

The limit counts executable dispatches: machine instructions and executable macros each consume one dispatch, and an `EX` subject consumes another dispatch when it executes. Thus `EX` plus its executed subject count as two dispatches. A configured maximum of zero means unlimited execution.

D.25 Self-Branch ABEND

A direct branch from an instruction to itself is detected as:

```
TSDS8029E
```

and terminates:

```
SYSTEM=EEE
REASON=8029
```

This is a narrow direct-self-branch check. BCT, BCTR, BXH, and BXLE are exempt from immediate self-branch ABEND handling because their register updates can make progress even when the target is the same instruction. Longer nonterminating loops, including exempt progress-branch loops that never terminate, are handled by the instruction limit.

D.26 WTO/WTOR ABENDs

WTO and WTOR are host terminal services under Linux and use `/dev/tty`.

Their runtime failures are fatal rather than recoverable R15 statuses.

Relevant diagnostics include:

Diagnostic	Meaning
8015	`/dev/tty` unavailable
8016	Terminal I/O failure
8017	WTO/WTOR message length invalid
8018	WTOR reply length invalid
8019	WTOR reply area invalid
8020	WTOR reply cannot be converted to EBCDIC

These generate SYSTEM=EEE ABENDs.

D.27 Record-I/O ABEND Diagnostics

Logical GET/PUT can generate fatal record-format diagnostics:

Diagnostic	Meaning
8025	Record exceeds permitted logical length
8026	Invalid variable-record RDW
8027	DCB/host/tape I/O failure

The detailed diagnostic text identifies the operation and data set involved.

D.28 Address and Protection Failures

Invalid or protected storage used by an executable operation is generally fatal.

The principal runtime address diagnostic is:

```
8002
```

Such failures can arise from:

- Invalid source or destination addresses.
- Access outside mapped virtual storage.
- Writes to LOWMEM.
- Data access to protected CODE storage.
- Invalid `$READ`/`$WRITE` buffers.
- Invalid PUT/GET buffers.

The operation normally terminates with SYSTEM=EEE and the applicable runtime diagnostic reason.

D.29 Runtime Diagnostic Summary

Important V67 runtime diagnostic numbers include:

Code	Meaning
8000	General runtime failure
8001	Invalid program counter/instruction address
8002	Invalid/protected runtime storage address
8003	Unsupported runtime operation
8004	Runtime output failure
8005	Arithmetic/runtime arithmetic error
8006	Save-stack underflow
8007	Save-stack overflow
8010	GETMAIN length error
8011	GETMAIN virtual-space failure
8012	GETMAIN host allocation failure
8013	FREEMAIN invalid address
8014	FREEMAIN length mismatch
8015	Terminal unavailable
8016	Terminal I/O failure
8017	WTO/WTOR message too long
8018	WTOR reply length invalid
8019	WTOR reply area invalid
8020	WTOR conversion failure
8021	DDNAME not allocated
8022	DCB/host/tape OPEN failure
8023	Invalid DCB/open state
8024	Invalid effective DCB attributes
8025	Record too long
8026	Invalid RDW
8027	DCB I/O/CLOSE failure
8028	Instruction limit reached
8029	Direct self-branch
8030	Assertion failed
8031	Assertion storage failure
8032	FREEMAIN interior address
8033	FREEMAIN already freed
8034	GETMAIN metadata failure

Appendix E provides the broader diagnostic-message catalog, including assembly-time diagnostic ranges.

D.30 Choosing Between RC Handling and ABEND Handling

A TSDSUTIL program should test R15 only for conditions that the service defines as recoverable.

Examples include:

```
OPEN TSDS8021
$SHOWDCB
$TESTDCB
$READ
$WRITE
GETMAIN RC
FREEMAIN RC
CLOSE recoverable conditions
GET EOF without EODAD
```

A program should not expect to regain control after a condition defined as fatal merely by checking R15 afterward. OPEN 8022/8023/8024, invalid GET/PUT storage, logical record-format errors, assertion failures, instruction-limit failures, and similar conditions terminate through the ABEND path.

This distinction is intentional: recoverable status represents an expected program decision point, while ABEND represents an invalid execution state or failure from which TSDSUTIL does not define continued processing.

D.31 Date/Time Macro Return Codes

`$VALDATE`, `$VALTIME`, `$CVTDATE`, `$DAYDIFF`, `$DAYADJ`, `$EXPDATE`, `$FMTDATE`, and `$TIMEADJ` share one date/time return-code namespace. A code has the same meaning wherever it can occur; not every macro returns every code.

R15	Meaning
0	Success
4	Invalid JDATE
8	Invalid GDATE
12	Invalid TIME
16	Invalid STIME/STCK/STCKE conversion value
20	Date/range or representation overflow
24	Invalid runtime operand or parameter
28	Nonexistent local civil time during DST transition
32	Ambiguous local civil time during DST transition
36	Output area too small
40	Invalid format string

Date/time macros preserve CC.

For date/time return-code failures, input values and effective addresses are captured before output commit, and outputs other than R15 remain unchanged. A direct register output `(R15)` is invalid. `0(R15)` is permitted as an indirect output address and uses the entry value of R15.

An invalid, unreadable, or protected virtual-storage reference is still handled by the normal TSDSUTIL address/protection runtime mechanism; it is not converted into a date/time value-validation return code.

D.32 Recoverable Date/Time RC Messages

In V89, a nonzero date/time macro RC produces a runtime listing warning by default, for example:

```
** TSDS8035W $FMTDATE returned RC=36: output area too small
```

The warning does not change R15 or any other program state. Use `#PRAGMA NO_RUNTIME_RC_MESSAGES` to suppress these warnings.

Appendix E — Diagnostic Messages

Applies to: TSDSUTIL Version 1.0.1

TSDSUTIL uses one four-digit diagnostic-code namespace. Severity is separate from the numeric code, so the same code is identified by its number while the suffix describes the severity of a particular occurrence.

A typical diagnostic is:

```
TSDS4001E Deferred expression could not be resolved in second pass: UNRESOLVED
```

The format is:

```
TSDSnnnnx
```

where `nnnn` is the four-digit diagnostic number and `x` is the severity suffix.

E.1 Severity Suffixes

Suffix	Severity	Meaning
I	Informational	Reports useful status; does not by itself flag a statement
W	Warning	Processing can continue, but the statement is flagged
E	Error	The statement is erroneous; processing status becomes error
S	Severe	A severe condition prevents normal processing or execution

Warning, error, and severe diagnostics count as flagged statements. Informational diagnostics do not.

The numeric code identifies the class of problem; the suffix is not part of the diagnostic number.

E.2 Listing Placement

First-pass assembly diagnostics are printed immediately after the affected source statement in the assembly listing.

Second-pass diagnostics are collected for the final **DIAGNOSTIC SUMMARY** rather than inserted into the earlier first-pass portion of the listing.

Runtime diagnostics can also be written into the listing when runtime listing output is enabled.

The diagnostic summary begins with either:

```
No Statements Flagged
```

or:

```
n Statements Flagged
```

before the individual diagnostic entries.

For COPY/INCLUDE translation units, a diagnostic can show both the global listing line and the originating source file/local line.

E.3 Message Text

The tables below describe the diagnostic **codes**, not one fixed literal message for every code.

Many diagnostics add context such as:

- Symbol name.
- DCB and DDNAME.
- PATH, DSN, and VOLSER.
- Effective RECFM/LRECL/BLKSIZE.

- Address and length.
- Allocation number and allocation source line.
- Source/include file.
- Specific invalid operand.

The contextual message should therefore be read together with the code.

For Version 1.0.1 runtime storage-operand failures in GET, PUT, `$READ`, and `$WRITE`, the contextual text includes the failing virtual `ADDRESS`, `LEN`, and the access reason (`Invalid Virtual Address` or `Write Protected`). The reason is intentionally the same classification used by TRACE.

TSDS4001E reports unresolved symbol names when they can be determined from the retained deferred expression trees. V93 recursively follows unresolved EQU dependencies to root missing symbols, deduplicates repeated references, and reports multiple root unresolved names in first-reference traversal order.

E.4 Diagnostic Ranges

The namespace is organized broadly by subsystem:

Range	Subsystem
1000–1999	API and engine setup
2000–2999	Source processing, translation units, COPY/INCLUDE, conditional assembly
3000–3999	Statement structure and assembly state
4000–4999	Symbols, expressions, relocation, fixups
5000–5999	Data definitions and DCB assembly
6000–6999	Instruction and operand assembly
7000–7999	Pragmas and configuration
8000–8999	Runtime execution
9000–9999	Internal engine failures

Ranges may be subdivided in future versions without renumbering established diagnostics.

E.5 1000–1999 — API and engine setup

Code	Description
TSDS1000	Invalid API argument or call parameter
TSDS1001	API version mismatch
TSDS1002	Memory allocation failure

E.6 2000–2999 — Source, translation units, INCLUDE/COPY, and conditional assembly

Code	Description
TSDS2000	Source input I/O failure
TSDS2001	Listing output I/O failure
TSDS2002	Physical source line too long
TSDS2003	Logical statement too long
TSDS2004	Invalid continuation
TSDS2005	#COPY processing failure
TSDS2006	Source/include nesting depth exceeded
TSDS2007	Recursive source inclusion
TSDS2008	#INCLUDE processing failure
TSDS2009	#INCLUDE used too late
TSDS2010	Translation-unit structure error
TSDS2011	Final linkage/translation-unit validation failure
TSDS2012	Conditional-definition error

Code	Description
TSDS2013	Conditional-assembly nesting depth exceeded
TSDS2014	Invalid conditional expression
TSDS2015	Invalid #IF/#ELIF/#ELSE/#ENDIF structure

E.7 3000–3999 — Statement structure and assembly state

Code	Description
TSDS3000	Empty statement where an operation is required
TSDS3001	Invalid label
TSDS3002	Missing operation
TSDS3003	Unknown operation
TSDS3004	Missing operand
TSDS3005	Unexpected operand
TSDS3006	Label required
TSDS3007	Label not allowed
TSDS3008	Operation invalid in the current assembly state
TSDS3009	Duplicate START
TSDS3010	No executable code

E.8 4000–4999 — Symbols, expressions, relocation, and fixups

Code	Description
TSDS4000	Duplicate symbol
TSDS4001	Undefined or unresolved symbol
TSDS4002	Expression error
TSDS4003	Invalid relocation
TSDS4004	Deferred fixup failure

E.9 5000–5999 — Data definitions and DCB assembly

Code	Description
TSDS5000	General data-definition error
TSDS5001	Invalid DS
TSDS5002	Invalid data type
TSDS5003	Invalid length
TSDS5004	Invalid repeat factor
TSDS5005	Invalid DC
TSDS5006	Initializer longer than destination
TSDS5007	Initializer numeric overflow
TSDS5008	Invalid initializer
TSDS5010	Invalid DCB
TSDS5011	Required DCB parameter missing
TSDS5012	Duplicate DCB parameter
TSDS5013	Invalid DCB parameter
TSDS5014	Invalid DDNAME
TSDS5015	Invalid RECFM

E.10 6000–6999 — Instruction and operand assembly

Code	Description
TSDS6000	Invalid instruction
TSDS6001	Invalid instruction operand
TSDS6002	Instruction operand out of range
TSDS6003	Invalid END operand
TSDS6004	Invalid END label
TSDS6010	Invalid address syntax
TSDS6011	Address/displacement out of range
TSDS6012	Invalid USING/address relationship
TSDS6013	Invalid address relocation
TSDS6020	Invalid SS1 operand syntax
TSDS6021	SS1 length out of range
TSDS6030	Invalid SS2 operand syntax
TSDS6031	SS2 length out of range
TSDS6040	Invalid macro syntax
TSDS6041	Invalid macro keyword
TSDS6042	Duplicate macro keyword
TSDS6043	Positional operand follows keyword operand

E.11 7000–7999 — Pragmas and configuration directives

Code	Description
TSDS7000	General #PRAGMA error
TSDS7001	Unknown #PRAGMA
TSDS7002	Invalid #PRAGMA operand/value
TSDS7003	#PRAGMA specified too late

V69 uses TSDS7002E for invalid storage-layout pragma operands, including sizes that are not positive whole-KiB quantities and completed DATA/CONTROL/LITERAL layouts that leave no GETMAIN space before CODE.

E.12 8000–8999 — Runtime execution

Code	Description
TSDS8000	General runtime failure
TSDS8001	Invalid program counter or instruction address
TSDS8002	Invalid or protected runtime storage address
TSDS8003	Unsupported runtime operation
TSDS8004	Runtime output failure
TSDS8005	Runtime arithmetic error
TSDS8006	$SAVE stack underflow
TSDS8007	$SAVE stack overflow
TSDS8010	GETMAIN length invalid
TSDS8011	GETMAIN virtual space unavailable
TSDS8012	GETMAIN host allocation failure
TSDS8013	FREEMAIN address is not a live allocation
TSDS8014	FREEMAIN length mismatch
TSDS8015	Linux /dev/tty unavailable
TSDS8016	Terminal I/O failure
TSDS8017	WTO/WTOR message length invalid
TSDS8018	WTOR reply length invalid
TSDS8019	WTOR reply area invalid
TSDS8020	WTOR reply conversion failure
TSDS8021	DDNAME not allocated
TSDS8022	DCB/host/tape OPEN failure
TSDS8023	Invalid DCB/open state

Code	Description
TSDS8024	Invalid effective DCB attributes OPEN diagnostics include the effective RECFM/LRECL/BLKSIZE and the specific validation rule that failed.
TSDS8025	Logical record too long
TSDS8026	Invalid variable-record RDW
TSDS8027	DCB/host/tape I/O or CLOSE failure
TSDS8028	Maximum instruction count reached
TSDS8029	Direct self-branch detected
TSDS8030	$ASSERT comparison failed
TSDS8031	$ASSERT storage unavailable or protected
TSDS8032	FREEMAIN address is inside a live allocation
TSDS8033	FREEMAIN allocation already freed
TSDS8034	GETMAIN allocation metadata failure
TSDS8035	Recoverable runtime service returned nonzero RC

E.13 9000–9999 — Internal engine failures

Code	Description
TSDS9000	Internal TSDSUTIL engine failure

E.14 TSDS4001 — Undefined or Unresolved Symbol

TSDS4001 is especially important because symbol resolution can be deferred until the second pass.

For example, a forward reference is legal if the symbol is eventually defined. If it is still unresolved after second-pass processing, TSDSUTIL reports TSDS4001E.

Earlier versions enhanced this diagnostic so unresolved symbol names could be recovered directly from the parsed expression tree. V93 extends that behavior through deferred EQU chains. The resolver first brings deferred EQU definitions to a fixed point; only expressions still deferred afterward receive TSDS4001E. The diagnostic walker then follows unresolved EQU fixups recursively to identify the root missing symbol or symbols when possible.

For example:

```
A         EQU   B+1
B         EQU   C+1
```

with `C` undefined produces diagnostics naming `C` as the root unresolved dependency. Repeated root names are deduplicated. Cycles are protected during recursive traversal so circular EQU dependencies cannot recurse indefinitely.

A typical message is:

```
TSDS4001E Deferred expression could not be resolved in second pass: C
```

E.15 Runtime Diagnostics and SYSTEM EEE

Many 8000-series runtime diagnostics are fatal.

For a TSDSUTIL-generated fatal runtime condition, the normal convention is:

```
SYSTEM=EEE
REASON=nnnn
```

where `nnnn` is the associated TSDS runtime diagnostic.

For example:

```
TSDS8028E Maximum instruction count exceeded
SYSTEM=EEE REASON=8028
```

Appendix D identifies which service conditions return an R15 status and which terminate through the ABEND path.

E.16 TSDS8021 Is Special During OPEN

Under V67, TSDS8021 is the only recoverable OPEN failure.

If the DCB's DDNAME is not allocated:

```
R15=8021
```

and execution continues.

Other OPEN failures, including TSDS8022, TSDS8023, and TSDS8024, cause SYSTEM=EEE ABENDs.

E.17 Diagnostic Codes Versus Service Return Codes

Not every value placed in R15 is a TSDS diagnostic number.

For example, physical tape `$READ` uses small service return codes such as:

```
0, 4, 8, 12, 16, 20, 24, 28, 32, 36
```

Those are `$READ` service statuses, not diagnostics named TSDS0004, TSDS0008, and so forth.

Conversely, conditional GETMAIN/FREEMAIN and the recoverable OPEN failure return an actual TSDS diagnostic number in R15.

Appendix D contains the service-specific return-code tables.

E.18 Virtual-storage diagnostics

V95 resolves the older diagnostic-number collision between instruction diagnostics and virtual-storage diagnostics. The storage/allocation conditions now use unique codes:

Code	Meaning
`TSDS6050`	Virtual-storage region/address-space full
`TSDS6051`	Storage/address error
`TSDS6052`	Storage-protection error

The established instruction diagnostics remain `TSDS6000` (invalid instruction), `TSDS6001` (invalid instruction operand), and `TSDS6002` (operand out of range). This preserves one shared four-digit diagnostic namespace without ambiguous numeric assignments.

E.19 Internal Diagnostic TSDS9000

TSDS9000 is reserved for failures inside the TSDSUTIL engine itself.

A user program should not normally depend on TSDS9000 as a recoverable condition. If it occurs, the surrounding message, listing, trace information, and reproducible input are important for diagnosing the engine failure.

E.20 Reading a Diagnostic Efficiently

When investigating a diagnostic, use these pieces together:

1. The four-digit code identifies the subsystem and general problem.
2. The suffix identifies severity.
3. The source/listing line identifies the triggering statement.
4. The contextual message identifies the particular symbol, DCB, address, operand, or resource.
5. For runtime failures, the ABEND reason normally repeats the four-digit TSDS diagnostic number.

For assembly problems, correct the earliest relevant diagnostic first. Later diagnostics can be consequences of an earlier syntax, symbol, or state error.

TSDS8035 - Recoverable runtime service RC

Severity: warning. A recoverable service completed with a nonzero return code. The message identifies the macro, RC, and common meaning. In V89 this is used by the date/time macro family. The RC remains in R15 and the normal macro error/atomicity rules still apply. `#PRAGMA NO_RUNTIME_RC_MESSAGES` suppresses this warning.

V90 date/time macro parser diagnostics

The date/time macro operand parser uses the existing macro-diagnostic family for structural errors: `TSDS6040E` malformed macro syntax, `TSDS6041E` invalid macro keyword, `TSDS6042E` duplicate keyword, and `TSDS6043E` positional operand after a keyword operand. A missing comma exposed by source continuation is therefore a 6040 macro-syntax error, not TSDS6001.

Expression-based `CL/XL/BL/PL` lengths that are unresolved, relocatable, zero, negative, or otherwise invalid produce `TSDS5003E`.

V91 macro value diagnostic

`TSDS6044E` identifies a syntactically well-formed macro keyword whose value or keyword relationship is invalid. V91 uses it for cases such as an invalid `$CVTDATE` type/time relationship or an invalid/overflowing `$TIMEADJ` adjustment expression. Structural errors remain 6040-6043.

Appendix F — Virtual Storage Layout

Applies to: TSDSUTIL Version 1.0.1

TSDSUTIL presents programs with a 24-bit virtual address space. The address space resembles a System/370 program's storage from the script's point of view, but it is managed by TSDSUTIL rather than being the address space of an emulated processor.

Valid virtual addresses range from:

```
000000 through FFFFFF
```

This appendix describes how TSDSUTIL divides that space and which parts a TSDSUTIL program can use.

F.1 Address-Space Overview

The current virtual-storage layout is:

```
000000  +------------------------------+
        | LOWMEM                       |
000FFF  +------------------------------+
001000  | DATA                         |
        +------------------------------+
        | CONTROL                      |
        +------------------------------+
        | LITERAL                      |
        +------------------------------+
        | GETMAIN                      |
EFFFFF  +------------------------------+
F00000  | CODE sentinel                |
F00002  | first possible instruction   |
        | CODE                         |
FFFFFE  | last even CODE address       |
FFFFFF  +------------------------------+
```

The exact boundary between DATA, CONTROL, LITERAL, and GETMAIN depends on the configured sizes of the lower fixed regions.

CODE always occupies the highest 1 MiB:

```
F00000-FFFFFF
```

F.2 24-Bit Addresses

TSDSUTIL addresses are 24-bit values.

The largest virtual address is:

```
FFFFFF
```

Registers are 32 bits, but an address used for TSDSUTIL storage or instruction addressing is interpreted under the language's 24-bit addressing rules.

For example:

```
        LA    R3,BUFFER
```

places the effective 24-bit virtual address of `BUFFER` into R3.

The virtual address is not a Linux host pointer.

F.3 LOWMEM — 000000 through 000FFF

The first 4096 bytes are LOWMEM:

```
000000-000FFF
```

LOWMEM is allocated, zero-filled, and **read-only** to the TSDSUTIL program.

This permits references to low addresses to behave predictably without allowing the program to modify the reserved low-memory area.

A read from valid LOWMEM returns its stored contents. A write to LOWMEM is a protection error.

User DATA therefore begins at:

`001000`

F.4 DATA

Ordinary `DC` and `DS` storage is allocated from the DATA region beginning at:

`001000`

The default DATA-region size is:

`4 KiB`

so with default configuration its initial range is:

`001000-001FFF`

Alignment rules can advance the DATA location counter before a definition is placed.

For example:

```
A         DS    C
B         DS    F
```

can leave alignment bytes between A and B.

DATA storage is readable and writable during execution.

F.5 CONTROL

CONTROL immediately follows DATA.

The default CONTROL size is:

`1 KiB`

This is engine-managed virtual storage rather than ordinary programmer-defined DC/DS storage.

A script should not assume that CONTROL addresses are available for application allocation merely because they lie numerically between DATA and GETMAIN.

With default sizes:

`002000-0023FF`

is CONTROL.

F.6 LITERAL

The LITERAL region immediately follows CONTROL.

The default literal-region size is:

`1 KiB`

Assembler-generated literals are allocated here.

For example:

```
          CLC   FIELD,=C'ABC'
          L     R3,=F'100'
```

references storage managed by the assembler in the LITERAL region rather than inserting ordinary DC definitions into the DATA stream after START.

With default region sizes, LITERAL initially occupies:

`002400-0027FF`

F.7 GETMAIN

The remainder of the lower address space after the fixed DATA, CONTROL, and LITERAL regions and before CODE is the GETMAIN region.

With default region sizes, GETMAIN therefore begins at:

`002800`

and extends through:

`EFFFFF`

GETMAIN obtains runtime dynamic storage from this region.

Example:

```
        GETMAIN R,LV=4096
```

returns the starting virtual address of the allocated area in R1.

The returned address is a TSDSUTIL virtual address, not a host pointer.

F.8 GETMAIN Allocation Lifetime

A GETMAIN allocation remains live until FREEMAIN releases it or execution ends.

FREEMAIN requires the exact allocation start and exact allocation size.

An address merely somewhere inside an allocation is not a valid FREEMAIN starting address.

TSDSUTIL tracks live allocations so that it can diagnose:

- Unknown allocation addresses.
- Interior addresses.
- Incorrect lengths.
- Double frees.

Conditional and unconditional GETMAIN/FREEMAIN behavior is described in Chapter 5 and Appendix D.

F.9 GETMAIN Initial Contents

New GETMAIN storage is zero-filled by default.

The debugging facilities can optionally request a one-byte fill value so newly allocated storage is easier to recognize during testing.

For example, a fill of:

`X'AA'`

causes the new allocation to contain repeated `AA` bytes.

This changes only the initial contents; it does not change the virtual-address layout.

F.10 CODE — F00000 through FFFFFF

CODE occupies the fixed upper megabyte:

`F00000-FFFFFF`

Unlike DATA storage, CODE is **not byte-addressable program storage**.

TSDSUTIL does not assemble System/370 machine-code bytes into this region. Instead, CODE addresses identify internal executable instruction objects.

This is one of the important differences between TSDSUTIL and a processor emulator.

F.11 F00000 Sentinel

Address:

`F00000`

is reserved as the CODE sentinel.

It is not an executable instruction.

The first possible instruction address is:

`F00002`

This makes zero-index/sentinel handling unambiguous internally while preserving a clean 24-bit CODE address range for the language.

F.12 Even Instruction Addresses

Executable instructions are assigned consecutive even addresses:

```
F00002
F00004
F00006
F00008
...
```

Each TSDSUTIL instruction therefore advances the logical CODE address by two.

This does **not** mean that every instruction is a two-byte System/370 machine instruction. The address is a logical instruction identifier.

For example, an MVC and an LR each consume one logical instruction position even though their real System/370 machine encodings would have different byte lengths.

F.13 CODE Is Not Machine Language

TSDSUTIL parses a machine-instruction statement into an internal executable representation.

For example:

```
        MVC   TARGET(10),SOURCE
```

is not converted into six bytes of System/370 object code at F00002.

Instead, F00002 identifies the internal MVC instruction object.

Consequently:

- There is no need for a program base register for CODE.
- CODE labels can be used directly as branch targets.
- Instruction addresses are logical TSDSUTIL addresses.
- The program cannot inspect the machine-code bytes of an instruction because no such bytes exist in virtual CODE storage.

F.14 No USING for CODE

TSDSUTIL does not require and does not permit System/370-style base-register establishment for executable CODE.

A branch target such as:

```
LOOP    DS    0H
        ...
        B     LOOP
```

uses the CODE label directly.

`USING` is reserved for DSECT addressing.

This differs from conventional System/370 assembly, where USING commonly tells the assembler which register can address a code or data location.

F.15 CODE Labels

A label associated with an executable statement receives that statement's even CODE address.

A permitted zero-length definition such as:

```
EOF     DS    0H
```

after START establishes an aligned logical CODE label without creating DATA storage.

No ordinary data allocation can resume after START.

F.16 CODE Protection

CODE is protected from ordinary program data reads and writes.

A storage instruction cannot treat a CODE address as if it were DATA:

```
        MVC   SOME_CODE_LABEL(4),BUFFER
```

is not a way to modify executable instructions.

Likewise, using a CODE address as an ordinary byte-data source is invalid.

TSDSUTIL therefore does not support self-modifying machine code.

The `EX` instruction is supported through TSDSUTIL's executable-instruction model; it does not require the script to rewrite CODE bytes.

F.17 DATA Address Versus D2(B2)

A named DATA symbol is one way to specify a storage address:

```
        L     R3,COUNT
```

An explicit displacement/base address is another:

```
        L     R3,0(R5)
        MVC   12(20,R6),0(R7)
```

In the latter form, the effective address is calculated from the displacement and register value.

For RX-style addressing:

```
D2(X2,B2)
```

the effective address is formed from the displacement plus the applicable index and base register values.

For storage-to-storage forms:

```
D1(L,B1),D2(B2)
```

the address components similarly identify virtual DATA storage.

A source label and a register/displacement expression can therefore identify the same virtual byte address by different means.

This use of a base register in an **operand effective address** is distinct from a System/370 assembler `USING` statement. TSDSUTIL allows register-based operand addressing while not requiring a CODE base register.

F.18 DSECT Addresses

A DSECT defines offsets and field metadata; it does not allocate an instance of the structure.

`USING` associates a DSECT with a register so named DSECT fields can resolve through that register.

For example, conceptually:

```
MYREC    DSECT ,
FIELD1   DS    CL8
FIELD2   DS    F
...
         USING MYREC,R5
```

means that FIELD1 and FIELD2 describe offsets from the structure instance whose runtime address is in R5.

This is the intended purpose of USING in TSDSUTIL.

F.19 Address Zero

Virtual address zero is part of LOWMEM, not a NULL host pointer.

It is therefore meaningful as a 24-bit virtual address value, although LOWMEM protection still governs what operations can be performed there.

Code should not confuse:

```
000000
```

with the absence of a host memory object.

F.20 Storage Crossing a Boundary

A storage operand must identify a valid accessible span, not merely a valid first byte.

If an operation requests several bytes, the entire requested range must satisfy the applicable storage and protection rules.

An operand beginning near the end of an accessible region cannot silently continue into CODE or beyond the 24-bit address space.

Invalid or protected runtime spans produce the applicable address/protection diagnostic and, where defined, a SYSTEM EEE ABEND.

F.21 Address Arithmetic

Registers are 32-bit, while virtual addresses are 24-bit.

Address-producing operations such as LA follow TSDSUTIL's 24-bit address behavior.

Programs should nevertheless keep the distinction clear:

```
register arithmetic value     32 bits
TSDSUTIL virtual address      24 bits
```

A register can contain values that are not valid accessible virtual-storage addresses.

The fact that a value fits in a register does not guarantee that it identifies valid DATA storage.

F.22 Default Layout

Using the default region sizes, the lower fixed layout is:

Region	Start	End	Default size
LOWMEM	000000	000FFF	4 KiB
DATA	001000	001FFF	4 KiB
CONTROL	002000	0023FF	1 KiB
LITERAL	002400	0027FF	1 KiB
GETMAIN	002800	EFFFFF	remainder
CODE	F00000	FFFFFF	1 MiB

Within CODE:

```
F00000    reserved sentinel
F00002    first possible executable instruction
...
FFFFFE    highest possible even instruction address
FFFFFF    inside CODE window but not an even instruction address
```

If configurable lower-region sizes are changed, DATA still begins at 001000 and CODE still begins at F00000; CONTROL, LITERAL, and GETMAIN boundaries move accordingly.

F.23 Configuring the Lower Regions

Beginning with V69, DATA, CONTROL, and LITERAL sizes can be changed with:

```
#PRAGMA DATA_SIZE 64K
#PRAGMA CONTROL_SIZE 4K
#PRAGMA LITERAL_SIZE 8K
```

The pragmas alter sizes only. LOWMEM, the DATA starting address, and CODE remain fixed. CONTROL and LITERAL move as necessary, and GETMAIN receives the remaining space through EFFFFF.

The completed layout must leave nonzero GETMAIN space before CODE. Invalid sizes or impossible layouts are rejected before ordinary program allocation begins.

F.24 Practical Consequences

The virtual-storage model leads to several useful rules for TSDSUTIL programs:

1. Ordinary user data begins at 001000.
2. LOWMEM can be read but not written.
3. DATA, literals, and GETMAIN storage use virtual byte addresses.
4. GETMAIN returns virtual addresses in R1.
5. CODE always resides in F00000-FFFFFF.
6. F00000 is never an instruction.
7. Instruction addresses are even and begin at F00002.
8. CODE is not byte-addressable machine code.
9. No CODE base register or CODE USING is required.
10. USING applies to DSECTs.
11. Register/displacement operands are still fully supported for data effective addresses.
12. CODE cannot be modified as ordinary storage.

These rules allow TSDSUTIL programs to retain familiar System/370 addressing techniques without requiring TSDSUTIL to emulate a physical System/370 processor or construct machine-language object code.

Appendix G — Complete Program Examples

Applies to: TSDSUTIL Version 1.0.1

This appendix contains complete or near-complete programs that combine facilities described elsewhere in the guide. Short feature demonstrations belong with the feature they explain; the examples here are intended as practical starting points.

Example	Principal facilities
G.1	Basic program structure and data
G.2	DSECT-based record layout
G.3	Linux logical input/output
G.4	Record selection and transformation
G.5	AWS standard-labeled logical I/O
G.6	BLP physical tape I/O and positioning
G.7	Translation-unit linkage
G.8	Executable regression/debugging

G.1 Basic Program

```
MESSAGE  DC    CL20'HELLO'
WORK     DS    XL20
*
         START ,
         MVC   WORK(20),MESSAGE
         $PRINT "WORK AREA:"
         $DUMP WORK,L'WORK
         $EOJ  0
         END   ,
```

Ordinary `DC` and `DS` definitions precede `START`. `L'WORK` keeps the diagnostic length synchronized with the storage definition. CODE addresses are assigned by TSDSUTIL; no CODE `USING` is required.

G.2 DSECT-Based Record Processing

```
RECORD   DS    XL16
*
MYREC    DSECT ,
NAME     DS    CL8
COUNT    DS    F
AMOUNT   DS    PL4
MYRECLEN EQU   *-MYREC
*
         START ,
         LA    R5,RECORD
         USING MYREC,R5
         MVC   NAME,=CL8'ALPHA'
         L     R3,=F'25'
         ST    R3,COUNT
         MVC   AMOUNT,=PL4'12345'
         $DFMT MYREC,RECORD
         $EOJ  0
         END   ,
```

`RECORD` is real backing storage defined outside the DSECT. `USING` associates R5 with that DSECT instance, and `$DFMT` interprets the storage using DSECT metadata without changing it.

G.3 Linux Logical Input and Output

```
#PRAGMA DD SYSIN,PATH='./input.txt',RECFM=FB,LRECL=80,BLKSIZE=800
#PRAGMA DD SYSPRINT,PATH='./output.txt',RECFM=FB,LRECL=80,BLKSIZE=800
```

```
*
SYSIN    DCB   DSORG=PS,MACRF=GM,DDNAME=SYSIN,EODAD=EOF
SYSPRINT DCB   DSORG=PS,MACRF=PM,DDNAME=SYSPRINT
RECORD   DS    CL80
*
         START ,
         OPEN  (SYSIN,INPUT)
         OPEN  (SYSPRINT,OUTPUT)
LOOP     GET   SYSIN,RECORD
         PUT   SYSPRINT,RECORD
         B     LOOP
EOF      DS    0H
         CLOSE (SYSIN)
         CLOSE (SYSPRINT)
         $EOJ  0
         END   ,
```

The application works with logical records. OPEN establishes the effective DCB attributes; GET and PUT perform the record processing described in Chapter 6.

G.4 Read, Select, Modify, and Write

This is the traditional TSDSUTIL utility pattern: read a record, inspect it, optionally change it, and write the result.

```
#PRAGMA DD OLD,PATH='./old.dat',RECFM=FB,LRECL=80,BLKSIZE=800
#PRAGMA DD NEW,PATH='./new.dat',RECFM=FB,LRECL=80,BLKSIZE=800
*
OLD      DCB   DSORG=PS,MACRF=GM,DDNAME=OLD,EODAD=EOF
NEW      DCB   DSORG=PS,MACRF=PM,DDNAME=NEW
RECORD   DS    CL80
*
         START ,
         OPEN  (OLD,INPUT)
         OPEN  (NEW,OUTPUT)
LOOP     GET   OLD,RECORD
*        If byte zero is X'01', clear bytes 20-23.
         CLI   RECORD,X'01'
         BNE   WRITE
         XC    RECORD+20(4),RECORD+20
WRITE    DS    0H
         PUT   NEW,RECORD
         B     LOOP
EOF      DS    0H
         CLOSE (OLD)
         CLOSE (NEW)
         $EOJ  0
         END   ,
```

Selection and transformation use ordinary System/370 instructions; TSDSUTIL supplies the data-set and execution environment.

G.5 AWS Standard-Labeled Logical Copy

```
#PRAGMA DD TAPEIN,TAPE='./input.aws',DSN='INPUT.DATA',VOL=SER001,+
             LABEL=SL,FILE=1
#PRAGMA DD TAPEOUT,TAPE='./output.aws',DSN='OUTPUT.DATA',VOL=SER002,+
             LABEL=SL,FILE=1
*
TAPEIN   DCB   DSORG=PS,MACRF=GM,DDNAME=TAPEIN,EODAD=EOF
TAPEOUT  DCB   DSORG=PS,MACRF=PM,DDNAME=TAPEOUT
RECORD   DS    CL80
```

```
*
         START ,
         OPEN  (TAPEIN,INPUT)
         OPEN  (TAPEOUT,OUTPUT)
LOOP     GET   TAPEIN,RECORD
         PUT   TAPEOUT,RECORD
         B     LOOP
EOF      DS    0H
         CLOSE (TAPEIN)
         CLOSE (TAPEOUT)
         $EOJ  0
         END   ,
```

OPEN processes the tape labels and establishes effective attributes. Blocking and unblocking remain service responsibilities, so the application uses the same GET/PUT model as a Linux-backed data set. See Chapter 7 for label and volume semantics.

G.6 BLP Physical Tape I/O and Positioning

Physical BLP processing works with tape blocks and tape-file boundaries rather than logical records.

```
#PRAGMA DD TAPEIN,TAPE='./input.aws',LABEL=BLP,FILE=1
*
TAPEIN   DCB   DSORG=PS,MACRF=R,DDNAME=TAPEIN
BUFFER   DS    XL32760
*
         START ,
         OPEN  (TAPEIN,INPUT)
*
*        Read one physical block.
         $READ TAPEIN,BUFFER,L'BUFFER
         LTR   R15,R15
         BNZ   READSTAT
         $DUMP BUFFER,64
*
*        Return to physical BOT, then position to the next file.
         $CONTROL TAPEIN,TYPE=REWIND
         LTR   R15,R15
         BNZ   CTLERR
         $CONTROL TAPEIN,TYPE=FSF
         LTR   R15,R15
         BNZ   CTLERR
*
         CLOSE (TAPEIN)
         $EOJ  0
READSTAT DS    0H
*        Handle tape-mark, EOI, or error status as required.
         CLOSE (TAPEIN)
         $EOJ  4
CTLERR   DS    0H
         CLOSE (TAPEIN)
         $EOJ  8
         END   ,
```

`$READ`, `$WRITE`, and `$CONTROL` use the shared physical-tape status model documented in Appendix D. `$CONTROL` supports BSB, BSF, FSB, FSF, and REWIND. A production program should distinguish the relevant boundary and error return codes rather than treating all nonzero values alike.

G.7 Separate Translation Units with ENTRY/VCON

A service translation unit can publish an entry point:

```
SERVICE  START ,
SERVFUNC ENTRY ,
*        SERVFUNC publishes the CODE location of the next statement.
         $RETURN
         END   ,
```

A main translation unit can include it separately and hold its linked address in a VCON:

```
#INCLUDE 'service.tsds'
SERVPTR  DC    V(SERVFUNC)
*
         START ,
*        SERVPTR is resolved during final TU linkage.
         $DUMP SERVPTR,L'SERVPTR
         $EOJ  0
         END   ,
```

Use `#COPY` instead when the included source should remain in the current translation unit. Chapter 8 is authoritative for namespace, ENTRY/VCON, and linkage rules.

G.8 Executable Regression and Debugging Program

```
RESULT   DS    F
*
         START ,
         L     R3,=F'20'
         A     R3,=F'22'
         ST    R3,RESULT
         $ASSERT R3,EQ,42
         $PRINT "TEST PASSED"
         $EOJ  0
         END   ,
```

`$ASSERT` makes expected state executable: failure causes a SYSTEM EEE ABEND and success has no program-visible side effects. TRACE can be enabled externally without changing the source; `$DUMP`, `$REGS`, `$SHOWDCB`, and `$DFMT` can be added when deeper inspection is needed. Chapter 10 describes the recommended debugging workflow.