

Realia COBOL 4.00 for DOS: Field Notes

Tara Stella



Realia COBOL 4.00 for DOS: Field Notes

Copyright © 2026 Tara Stella

E-mail: tara@tara.sh

Website: <https://tara.sh/>

Version 1.0, October 2026

This is an independent guide based on direct testing of Realia COBOL 4.00 and consultation of period Realia and CA-Realia documentation. It is not an official Realia or Computer Associates publication.

This document may be copied, mirrored and redistributed freely, in its entirety and without modification, without further permission from the author.

Excerpts and extracted text may also be reproduced, provided that the document and its author are clearly credited.

The source-code examples included in this document are licensed under the BSD Zero Clause License. See Appendix F.

Table of Contents

Preface.....	5
About these notes.....	6
The Realia COBOL Environment.....	8
The compiler and the toolchain.....	8
Compiler Error Levels.....	9
Source Format and Paragraph Names.....	9
Other Realia Development Tools.....	10
Terminal User Interfaces.....	12
Screen Section.....	12
Keyboard and function keys handling.....	14
File Handling.....	17
Sequential Files.....	17
Physical DOS Format and GnuCOBOL Conversion.....	18
Deleting and Renaming Files.....	19
Sorting Files.....	21
Calling and Structuring Programs.....	23
Dialect Limits and Surprises.....	25
Reference Modification Is Not Supported.....	25
COMP-X Is Rejected.....	26
Report Writer.....	27
The Finished Application.....	28
Appendix A: FIELD-TERMINATOR Reference.....	30
Appendix B: ACCEPT_CONTROL Reference.....	33
Appendix C: Indexed File Organization.....	36
Appendix D: An Implicit-Scope Nesting Trap.....	38
Appendix E: Sources and References.....	39
Appendix F: Example Programs.....	41
HELLO.COB: Minimal build.....	41
SCREEN.COB: Screen and keys.....	42
SEQFILE.COB: Sequential files.....	44
SORTDEMO.COB: Native sort.....	46
LICENSE for the source files.....	48

Preface

While carrying out a broader piece of research, I found myself looking for a COBOL compiler for DOS. A few are still somehow available on the Internet, including Microsoft COBOL and, if you are lucky, some versions of Micro Focus COBOL, both much more mainstream choices. Then I stumbled across Realia COBOL 4.00 for DOS. What first caught my attention was how remarkably small it was: in less than a megabyte, you have the compiler, linker and libraries. The lack of documentation probably should have discouraged me. Instead, I got stubborn and decided that I wanted to know more about it.

The experiment gradually turned into a small book-library management application, written entirely in Realia COBOL 4.00, with add, browse, search, edit, delete and sort functions. The point was not to exercise every corner of the compiler, but to find out whether it could still be used to build the kind of small, full-screen TUI application I had in mind. In particular, I wanted something based around field-oriented interaction and function keys, while keeping the data itself in simple sequential files that could remain understandable and portable outside Realia.

What started as a small experiment ended up occupying the better part of three months, with enough dead ends, isolated tests and compiler archaeology along the way that the notes gradually became a project of their own.

This is adjacent to a previous personal experiment described in my essay *Cobolito/400: A Tiny Data Appliance in the Lineage of the IBM 5280*.¹ The two projects are not the same, but they come from much the same place: an interest in small, understandable applications, text interfaces, and data that does not become hostage to the program that created it.

Useful documentation for this particular DOS release turned out to be remarkably difficult to find. Much of what surfaces easily online refers instead to substantially later CA-Realia products, and several things described there simply do not behave the same way on the DOS 4.00 compiler. Since I had already accumulated rather a lot of notes while finding this out the hard way, it seemed useful to turn them into something that somebody else might actually be able to read.

These are therefore not intended to be a complete Realia COBOL manual, nor a COBOL tutorial. I assume the reader already has a basic knowledge of COBOL and some familiarity with GnuCOBOL or another compiler. This is instead an attempt to document

1 ¹ *Cobolito/400: A Tiny Data Appliance in the Lineage of the IBM 5280*, available at <https://www.tara.sh/pdf/cobolito400-2026-07.pdf>

the parts of Realia COBOL 4.00 that mattered while building a real, if deliberately small, application: what worked, what did not, what behaved differently from more familiar implementations, and which traps consumed rather more tea than they deserved.

About these notes

The compiler discussed here is specifically Realia COBOL 4.00 for DOS, and everything should be read with that version in mind. Most compilation and development took place under DOSBox. The resulting application was also tested under FreeDOS in a virtual machine, as well as under dosemu2 and emu2. The aim was never a systematic audit of every feature in the compiler. The perimeter was intentionally much smaller: build a practical TUI-oriented COBOL application using sequential files for portable data, and investigate the compiler as far as that required.

During development, the primary source of truth was the compiler itself. Where behaviour could not be explained from source-level testing alone, I also inspected symbols and embedded strings in the compiler and runtime libraries to identify internal routine names, diagnostics and other clues about how the environment was put together. Findings were normally established through an actual successful compile and run, or through the compiler's own ".LST" listing when it stated something directly, for example that a particular verb was not supported by that version. This matters because documentation from other Realia generations sometimes turned out to describe facilities that either were not present in 4.00 or behaved differently enough to send the investigation in entirely the wrong direction.

Only after the book application was already complete did I obtain, with the help of Eugenio Di Lorenzo, remarkably rare copies of the original Realia COBOL 3.00 manual and the Realia COBOL 4.2 User Guide for DOS and OS/2. They therefore served mostly as retrospective evidence rather than instructions used to build the application. They allowed the experimental findings to be checked against documentation from the closest contemporary releases before and after the version tested here, and in many cases confirmed what had already been discovered by trial and error. Where the manuals and Realia COBOL 4.00 disagree, I have tried to make that distinction explicit rather than silently assuming that behaviour documented for another release must also apply here.

The version numbers deserve a little care. The original Realia COBOL 3.00 manual is documentation for the earlier DOS compiler and should not be confused with the much later CA-Realia II Workbench 3.0 and 3.2 documentation that is considerably easier to find online. Those Workbench releases belong to a different product generation. They

were useful during the investigation, particularly when nothing closer was available, but they are not authoritative documentation for the DOS 4.00 compiler.

Everything in these notes still comes from one person's hands-on experimentation with a single copy of Realia COBOL 4.00, organised after the fact. It is not a systematic audit and may not hold for every installation, patch level or obscure edge case. Some details may be incomplete, and some may eventually turn out to be wrong when somebody with deeper knowledge of the compiler comes along. Treat this as a practical starting point for your own testing, not a replacement for it.

The Realia COBOL Environment

The compiler and the toolchain

The Realia COBOL 4.00 for DOS distribution I was able to obtain is remarkably small. Ignoring the bundled Microsoft EDIT.EXE, which is simply an editor and not part of the Realia toolchain itself, the archive essentially consists of:

LINK.EXE
LINK.FIG
REALCOB.AUT
REALCOB.EXE
REALCOB.LIB
REALCOB.MAP
REALDOS.LIB

REALCOB.EXE is the actual COBOL compiler. It compiles source files into standard object files but does not produce an executable by itself. A simple one-line compilation looks like this:

```
REALCOB BOOKADD.COB,BOOKADD.OBJ,BOOKADD.LST
```

The two runtime libraries, REALCOB.LIB and REALDOS.LIB, are byte-identical Microsoft OMF-format object libraries, so only one of them needs to be linked into the final executable.

Linking is a separate step performed by LINK.EXE. This is not a Realia-proprietary linker, but a bundled copy of OPTLINK, the general-purpose DOS linker originally produced by SLR Systems. In other words, Realia COBOL produces conventional DOS object files and then uses an ordinary third-party linker to turn them into an executable.

A minimal link command is:

```
LINK BOOKADD.OBJ,BOOKADD.EXE,,REALDOS,,
```

The compiler also produces a .LST listing file. This turned out to be the most reliable source for compiler diagnostics, particularly under DOSBox where terminal scrollbar can easily lose the actual error that matters. When something goes wrong, the listing is usually the first place worth looking.

Compiler Error Levels

Once the compilation process is wrapped in a BAT file, the compiler's return codes become rather more useful than they first appear. Typing the REALCOB command and then the separate LINK command by hand is tolerable once or twice. Doing it every time a multi-module application is rebuilt becomes old rather quickly.

The error levels I confirmed empirically for REALCOB.EXE are:

0 = clean compile

2 = warnings only

6 = compilation error

This is enough to distinguish a successful compile with warnings from one that should stop a build before the linker is invoked.

There is one DOS-era wrinkle. In the environment I used, %ERRORLEVEL% is not available as an expandable batch variable. The portable older-DOS mechanism is instead the special IF ERRORLEVEL n conditional, which means “n or higher”, not “exactly n”.

For example, a build script can allow warning-only compilations to continue while stopping on real errors:

```
REALCOB BOOKADD.COB,BOOKADD.OBJ,BOOKADD.LST
IF ERRORLEVEL 3 GOTO COMPILE_ERROR

LINK BOOKADD.OBJ,BOOKADD.EXE,,REALDOS,,
```

Because the warning level is 2 and the confirmed error level is 6, testing for ERRORLEVEL 3 cleanly separates the two cases without needing to know the exact value.

Source Format and Paragraph Names

Realia COBOL 4.00 expects traditional fixed-format COBOL source. Column 7 is the indicator column, normally blank or “*” for a comment, while the program text begins in column 8 using the usual Area A/Area B layout. Source lines are effectively limited to column 72. Anything extending beyond it risks being silently truncated or misparsed. This is easy to hit with longer literals or screen descriptions. In this project, even the line containing the function-key legend eventually ran into the limit.

There is another small trap if, like me, you are editing the source under Linux. In the environment tested here, Realia expects DOS-style CRLF line endings. A source file containing plain Unix LF line endings can be interpreted as one enormous physical line,

resulting in a spectacular cascade of syntax errors having very little to do with the program you actually wrote.

Realia also strongly expects executable statements in the *PROCEDURE DIVISION* to sit inside a named paragraph, although it does not strictly require this to compile or run. A test program containing bare *DISPLAY*, *PERFORM*, *END-PERFORM* and *STOP RUN* statements immediately below *PROCEDURE DIVISION.*, with no paragraph name at all, still compiled and executed correctly. The compiler did, however, produce warning 0346 for every top-level statement:

```
0346 W 'DISPLAY' In invalid area; assumed valid
```

The important part is “assumed valid”: these are warnings rather than semantic failures, and the generated program behaves as intended. They are also easily avoided. A single paragraph name, even something as simple as *MAIN.*, is enough to anchor the entire body and silence the warnings. There is no need to invent a forest of paragraphs merely to keep the compiler happy.

Other Realia Development Tools

The broader Realia development environment also included tools beyond the compiler and linker. The Realia COBOL 3.00 distribution available to me contains REALDEBUG.COM, an interactive COBOL debugger, and RED.COM, the Realia editor. Neither is present in the surviving 4.00 archive used for this project, and I have not tested either with the 4.00 compiler, so they should not be taken as confirmed components of that particular distribution.

REALDEBUG.COM identifies itself as “Realia REALDEBUG V3.14” and “Interactive COBOL Debugger”. Eugenio Di Lorenzo, who used Realia professionally at the time, recalls using it daily and describes it as a complete and powerful debugger/animator integrated into the Realia environment.

RED.COM identifies itself as version 3.10 and is the Realia source editor. Some surviving descriptions call it a “screen editor”, but in practice this refers to its full-screen editing interface rather than to any facility for designing SCREEN SECTION layouts. I briefly tested the copy included with the Realia COBOL 3.00 distribution. It is an enhanced COBOL-oriented source editor with conveniences such as listing-file integration, but it does not provide an alternative to the SCREEN SECTION and ACCEPT mechanisms described elsewhere in these notes. The documentation included with the Realia 3.00 distribution describes integration with compiler listing files, including an Alt-E command that opens the most recent listing and positions the editor at source lines containing errors.

Both utilities are interesting parts of the surrounding Realia environment, but they remain outside the directly tested 4.00 toolchain described in these notes.

Terminal User Interfaces

For the book-library management application, I wanted an interaction model close to the one I was already familiar with from IBM i and had previously explored in the CrumbStation example described in my essay. The application should use a full-screen text interface, with proper field handling and function keys assigned to specific actions rather than behaving like a sequence of prompts.

I deliberately chose function keys because I like that style of interface, but also because they belong naturally to the IBM i and midrange world: visible actions, clear keys and very little ambiguity. There is something honest about a screen that tells you what the keys do instead of hiding the work behind a floating cloud of gestures, menus and hope.

Screen handling, however, is one of those areas where COBOL implementations tend to expose the personality of the environment underneath them. Realia COBOL has a *SCREEN SECTION* and statements that look superficially familiar if you have used GnuCOBOL, but the similarities only go so far. Getting from a screen that merely compiled to one with reliable field handling and function-key navigation took a considerable amount of experimentation, helped eventually by a few clues found in contemporary Realia source code.

Screen Section

A basic *SCREEN SECTION* in Realia COBOL looks familiar enough if you have used screen handling in another COBOL implementation:

```
SCREEN SECTION.

01  ENTRY-SCREEN.
    05  BLANK SCREEN.
    05  LINE 2 COLUMN 2 VALUE "Name (F1=quit)          : ".
    05  SCR-NAME LINE 2 COLUMN 26 PIC X(20) USING WS-NAME.
    05  LINE 4 COLUMN 2 VALUE "Age  (F2=skip)          : ".
    05  SCR-AGE  LINE 4 COLUMN 26 PIC 99    USING WS-AGE.

01  RESULT-SCREEN.
    05  LINE 7 COLUMN 2 VALUE "You entered:".
    05  LINE 8 COLUMN 2 PIC X(20) FROM WS-NAME.
    05  LINE 9 COLUMN 2 PIC 99    FROM WS-AGE.
```

The first important detail is the relationship between *PICTURE* and the data-transfer clauses. A screen item with a *PIC* clause must also specify *FROM*, *TO* or *USING*. A bare *PIC* without one of those directions is rejected outright. The reverse case is slightly more treacherous. In my tests, omitting an explicit *PIC* from a *FROM* item did not necessarily

produce a useful compile-time error. Instead, the field could be rendered with the wrong width or otherwise garbled because Realia did not reliably infer the screen picture from the linked Working-Storage item. The problem was isolated directly with *FROM*. Throughout the rest of the project I therefore treated an explicit matching *PIC* as mandatory for *FROM*, *TO* and *USING*, which proved to be a safe working rule.

The three clauses themselves have distinct roles. *TO* is input-only: data entered during an *ACCEPT* is transferred from the screen item into the linked variable. *FROM* is output-only and supplies the value used during a *DISPLAY*. *USING* is bidirectional, allowing the same field to be both displayed and accepted. This may be obvious if you already know this style of *SCREEN SECTION*, but it caught me early on because most of my previous experience had been with GnuCOBOL and I had developed slightly different habits around screen fields.

There is another distinction worth making explicit: the screen item's own name is not a live alias for the Working-Storage variable linked through *USING*. Moving a value directly into the screen item does not alter the value that a later *DISPLAY* obtains from the underlying variable. If the program wants to change what the field will display, it should move the value into the linked Working-Storage item itself.

USING worked correctly with every field width exercised in this project, including fields of 50 and 60 characters. For example:

```
05 LINE 7 COLUMN 5 VALUE "Title      :".  
05 SCR-TITLE LINE 7 COLUMN 16 PIC X(50) USING WS-TITLE.
```

The names such as *SCR-TITLE* also become important once keyboard handling is introduced. Rather than accepting an entire screen at once, the application accepts individual elementary screen items, and those names provide the targets for the individual *ACCEPT* statements. This is the basis of the function-key handling described in the next section.

One of the more inconvenient limitations appears when trying to build browse or list screens. A *SCREEN SECTION* item in this compiler cannot reference a subscripted identifier. A construct such as *FROM SOME-TABLE(SOME-INDEX)* is rejected with errors beginning with “Unrecognizable word or literal '(', even though the same subscripted identifier works normally in the *PROCEDURE DIVISION*. There is also no usable *OCCURS* mechanism for repeating screen rows. A browse screen therefore has to be physically unrolled, with one screen description for each visible row.

The practical workaround is to keep the real data in an ordinary subscripted table and introduce flat, non-subscripted bridge variables for the screen. Before a *DISPLAY*, the program copies each visible row from the table into those bridge fields. After an *ACCEPT*,

any editable values can be copied back again. It is certainly more verbose than being able to reference the table directly, particularly for a twelve-row browse screen, but it keeps the awkwardness confined to the screen interface while the rest of the program can continue using normal tables and subscripts.

Finally, there is one genuine compiler crash worth knowing about. A floating-insertion numeric-edited *PICTURE*, such as *PIC ----9*, used on a *FROM* screen item causes an internal compiler failure rather than a source diagnostic:

```
*** Pass 3 fatal error, internal error code = 0522  
Internal logic error - Inform Realia Corp.
```

This was confirmed by direct bisection. Replacing *PIC ----9* with the fixed-width *PIC 9999*, while leaving the rest of the program unchanged, compiled cleanly. Fixed numeric-edited pictures such as *PIC 9999* and *PIC ZZZ9* also worked normally elsewhere in the project. I therefore treat floating-insertion pictures as unsafe inside the *SCREEN SECTION* on this compiler unless proven otherwise. Only the *----9* form was tested directly, so similar forms such as *++++9* or *\$\$\$\$9* should be considered suspect rather than declared broken outright.

Keyboard and function keys handling

Working out keyboard interaction, and function keys in particular, was probably one of the most time-consuming parts of this experiment. Some of the difficulty came from approaching Realia with expectations formed by other COBOL implementations, in my case mostly GnuCOBOL. The statements looked familiar enough that I initially assumed the interaction model would be familiar too. It was not.

The mistake that consumed most of the time was accepting the entire screen group at once. Realia provides its own navigation when a full screen is accepted in this way, but that is not a particularly useful model if the application wants to control navigation itself and dispatch specific function keys to application actions. The turning point came before I had access to the original manuals, when I found a 1992 CA-authored sample program that I obtained as *REALIA-SCRADDR.CBL* (PROGRAM-ID *SCRADDR*). Rather than accepting an entire screen, it accepts one elementary screen item at a time and inspects the special register *FIELD-TERMINATOR* after each *ACCEPT*.

In practical terms, this means that the named screen items introduced in the previous section are not merely labels for readability. They become the individual targets of *ACCEPT*, allowing the application to decide what should happen after each field. *FIELD-TERMINATOR* reports the key that ended that particular *ACCEPT*, and can be compared directly against numeric constants declared as *PIC S9(4) COMP-5*. F1 through F10, for

example, are returned as values 59 through 68. The complete table is included in the Appendix A.

A simplified example continuing the screen shown previously looks like this:

```
MAIN.  
  DISPLAY ENTRY-SCREEN.  
  
  PERFORM UNTIL WS-QUIT = "Y"  
    ACCEPT SCR-NAME  
    IF FIELD-TERMINATOR = 59  
      DISPLAY SPACE WITH BLANK SCREEN  
      DISPLAY "Cancelled with F1 - nothing else shown."  
      MOVE "Y" TO WS-QUIT  
    ELSE  
      ACCEPT SCR-AGE  
      IF FIELD-TERMINATOR = 60  
        MOVE 0 TO WS-AGE  
      END-IF  
      DISPLAY RESULT-SCREEN  
      MOVE "Y" TO WS-QUIT  
    END-IF  
  END-PERFORM.
```

Once input is handled field by field, the application also becomes responsible for navigation that a group *ACCEPT* would otherwise provide automatically. This is slightly more work, but it also gives considerably more control. In the book-management application, both Down and Tab advance to the next field, while Up and Back-Tab move to the previous one. Function keys can interrupt the same input loop and invoke actions without first completing the entire form. That produced much more closely the kind of TUI interaction I had been aiming for.

There is a complication around *CRT STATUS*. During the original investigation I tried to use it with a group *ACCEPT* and could never obtain useful function-key information from it. The later 4.2 manual, however, explicitly documents *CRT STATUS* for the relevant forms of *ACCEPT*, so I would not take that failed experiment as evidence that the feature itself is unsupported in 4.00. The original test may simply not have configured *SPECIAL-NAMES* correctly.

In any case, *CRT STATUS* would not have solved the real problem. Its three-byte result describes the broad type of event, such as a terminator or function-key condition, but does not identify which function key was actually pressed. An application that needs to distinguish F1 from F3 or Esc still needs *FIELD-TERMINATOR*. For this project, individual field *ACCEPT*s plus *FIELD-TERMINATOR* therefore became the simpler and more useful model, regardless of why the earlier *CRT STATUS* experiment failed.

Another small issue appeared while implementing the single-character Edit/Delete selectors in the browse screen. I wanted an action to occur only after the operator deliberately pressed Enter rather than immediately after typing the one character that completely filled the field. This led to testing the *AUTO* clause, also documented as *AUTO-SKIP*. The behaviour is genuinely opt-in: a field carrying *AUTO* advances as soon as its final character is entered, while an otherwise identical field without it continues waiting for a terminator such as Enter. An isolated three-field test confirmed this directly, including both one-character and wider fields. In other words, merely filling a field does not implicitly terminate its *ACCEPT*.

The same investigation also uncovered the callable *ACCEPT_CONTROL* routine, which changes several aspects of subsequent *ACCEPT* behaviour, including automatic entry, cursor-key handling and treatment of empty fields. The book-management application did not ultimately need it, but I tested it while investigating the keyboard model and have kept the details in the Appendix B for completeness.

File Handling

File handling is another area where otherwise portable COBOL starts to depend on the environment around the compiler. The language gives us familiar concepts such as sequential, relative and indexed files, but the physical representation on disk, runtime conventions and implementation-specific extensions still belong to the particular COBOL system underneath them.

For this project that mattered more than usual. One of the design goals was to keep the data understandable outside the application that created it. Realia's indexed files work, as explored later in the Appendix, but they use their own implementation-specific format. Sequential files looked like the more deliberately boring choice: less convenient in places, certainly, but much easier to inspect, convert and carry elsewhere.

Sequential Files

The book-library application therefore uses sequential files for its primary data. They are not especially clever, and operations such as modifying or deleting a record require more work than they would with an indexed organization, but that simplicity is part of the attraction. The records remain close to ordinary text and can be moved between systems without requiring the original Realia runtime merely to understand what is in the file.

Realia COBOL 4.00 for DOS accepts *SEQUENTIAL*, *RELATIVE* or *INDEXED* organization in the *FILE-CONTROL* section. For a DOS disk file, plain *ORGANIZATION IS SEQUENTIAL* produces a text-oriented DOS file. Its exact physical representation is described below. There is no *LINE SEQUENTIAL* organization keyword on this compiler. On implementations such as GnuCOBOL, Micro Focus or Fujitsu, that qualifier is commonly used to distinguish text-oriented records from other sequential file representations. Under DOS, Realia's ordinary sequential files already occupy that role.

A minimal declaration looks like this:

```
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT SAMPLE-FILE  
        ASSIGN TO "SEQFILE.DAT"  
        ORGANIZATION IS SEQUENTIAL  
        FILE STATUS IS WS-FILESTATUS.
```

Physical DOS Format and GnuCOBOL Conversion

Realia sequential files are text files, but “text file” is not quite the whole story. If the data is intended to move between Realia and another system, it is useful to understand the exact physical representation the runtime writes to disk.

Records are normally terminated with CR/LF. On the final record, however, the runtime writes CR followed directly by the DOS Ctrl-Z EOF marker, without a final LF.

When writing a sequential file, the Realia runtime compresses runs of blank padding into tab characters, using the usual eight-column tab stops, rather like the Unix *unexpand* utility. This was confirmed by examining generated files with *cat -vte*: fields that should have contained fixed-width space padding instead contained tabs, while the final record ended in CR followed by Ctrl-Z rather than CR/LF.

This made a great deal of sense in the environment the compiler came from. Replacing long runs of spaces with tabs could save a surprising amount of disk space when records contained many padded fields. It is clever, but it also means that a Realia sequential file is not immediately interchangeable with the fixed-width text file another compiler may expect.

To read a Realia file with GnuCOBOL, IBM i, or an ordinary Unix tool, the tabs therefore need to be expanded back into spaces, the DOS line endings converted to the host convention, and the trailing Ctrl-Z removed. One possible conversion under Unix is:

```
dos2unix FILE.DAT
tr -d '\032' < FILE.DAT > FILE.tmp
expand -t8 FILE.tmp > FILE.dat
```

The *-t8* is significant: Realia's compression follows eight-column tab stops, so expanding with the same width reconstructs the original fixed-width record layout.

This direction was tested rather than inferred only from inspecting the file. I wrote a small GnuCOBOL program using *LINE SEQUENTIAL* and the same record definition as the Realia test program. After converting the Realia-generated file as shown above, GnuCOBOL read the records correctly, including the original fixed-width field boundaries. In the test case the reconstructed record was 34 bytes wide. This provided a useful practical check that expanding the tabs was actually restoring the data layout rather than merely producing something that looked plausible in a text editor.

The useful part is that the conversion is not inherently one-way. A small DOS application may just as easily be the consumer rather than the producer of the data. For example, a point-of-sale or inventory application could receive a fixed-width sequential export

generated by GnuCOBOL or IBM i, convert it into the representation expected by the DOS environment, and then use it locally.

To reproduce Realia's on-disk representation from an ordinary fixed-width Unix text file, the reverse transformation is:

```
unexpand -a -t8 FILE.dat > FILE.tmp
unix2dos FILE.tmp
truncate -s -1 FILE.tmp
printf '\032' >> FILE.tmp
```

The blank padding is first compressed using eight-column tab stops and the Unix line endings are converted to CR/LF. The final LF is then removed and replaced by the DOS Ctrl-Z EOF marker, leaving the last record terminated by CR followed directly by Ctrl-Z, matching the representation produced by Realia itself. This recipe assumes that the Unix input file ends with a normal LF. Without it, the truncate step would remove the final data byte instead.

This direction was also tested directly. A fixed-width Unix CrumbStation data file was converted into this DOS representation and then read by a small program compiled with Realia COBOL 4.00 using an ordinary sequential file. The test program read all three records correctly, reported no misaligned records, and finished with file status 00.

The conversion is therefore usable in both directions: Realia-generated sequential files can be converted into ordinary fixed-width text for use by other systems, while fixed-width text generated elsewhere can be converted back into the physical representation accepted by Realia.

There is also a small Unix-side trap. `wc -l` counts LF characters rather than logical records. Because a Realia file ends its final record with CR followed directly by Ctrl-Z, that last record contributes no LF and `wc -l` reports one record fewer than the file actually contains.

Deleting and Renaming Files

The book-management application allows records to be edited or marked for deletion from the browse screen. A sequential file, however, is not a database table, nor does it provide the direct record replacement and deletion mechanisms available with indexed files. There is no tiny trapdoor underneath a record where the program can quietly make it disappear or alter it in place. For this application, editing and deletion therefore mean rebuilding the file.

The usual approach is simple: read the original sequential file from beginning to end and write a temporary replacement. Records being kept are copied unchanged, edited records

are written with their replacement contents, and deleted records are simply omitted. Once both files have been closed, the temporary file can replace the original.

The first method I used relied on a Realia extension I discovered before obtaining the original manuals. While inspecting strings and compiler vocabulary during the investigation, I found the statement *DELETE FILE* and then confirmed it experimentally. It is a genuine Realia extension rather than standard ANSI COBOL, and removes a disk file provided that the file is not currently open.

Using only that facility, the replacement sequence is straightforward: write the rebuilt contents to a temporary file, close both files, use *DELETE FILE* on the original, recreate it with *OPEN OUTPUT*, copy the temporary contents back, and finally delete the temporary file. It works, although copying the complete file a second time is slightly inelegant when the temporary file already contains exactly what is wanted.

A cleaner alternative is to rename the temporary file into place. I had used the same general technique in earlier GnuCOBOL experiments, but did not initially know that Realia provided a DOS rename interface. The facility is not exposed as COBOL syntax. Instead, Realia provides the callable routine *DOS_RENAME_FILE*:

```
CALL "DOS_RENAME_FILE" USING OLD-NAME NEW-NAME
```

I confirmed this directly, and it is also documented among the DOS-interface routines in the later 4.2 manual. Internally it wraps the DOS rename/move service, INT 21h function 56h. It can rename a file or move it between directories on the same drive, but not between drives.

The return value is supplied through *RETURN-CODE*:

Return value	Meaning
---------------------	----------------

0	success
2	file not found
3	path not found
5	access denied
17	not same device

Both arguments are PIC X(79) fields containing ASCIIZ strings rather than ordinary space-padded COBOL text. In practical terms, the filename must be followed by a zero byte. Realia's documentation uses LOW-VALUE for that terminator:

A typical construction is:

```
STRING FILE-NAME DELIMITED BY SPACE  
      LOW-VALUE DELIMITED BY SIZE  
      INTO ASCIIZ-FIELD
```

LOW-VALUE is the COBOL figurative constant for the lowest value in the collating sequence, which in this ASCII DOS environment gives the required 00h terminator. Anything after that byte in the destination field is ignored.

For the rebuild strategy, the practical sequence can therefore be reduced to: write the new version to a temporary file, close everything, remove the old file, and rename the temporary file into its place. The application still performs a complete sequential rebuild, but it avoids copying the reconstructed data yet again simply to recover the original filename.

Sorting Files

Another consequence of using sequential files is that records naturally remain in the order in which they were written. For an application that mostly appends new records, this means that a browse screen eventually reflects insertion order rather than anything particularly useful to the person looking at it.

Fortunately, COBOL already provides the machinery needed to reorganise the file. Realia COBOL 4.00 supports the *SORT ... USING ... GIVING ...* form cleanly for a simple whole-file sort. The operation is not performed in place: *USING* identifies the input file, while *GIVING* writes the sorted records to a separate output file.

The input and output files should not already be open when *SORT* is executed. Realia handles opening, reading, writing and closing them as part of the sort operation itself. Once the sorted file has been produced, it can replace the original using the same file-management techniques described in the previous section.

In the book-management application I also added a backup step before sorting anything. This is simply another sequential read/write copy loop, creating a copy of the original file before the sort and replacement take place. It is hardly sophisticated, but when the entire catalogue is about to be rewritten, a little deliberate cowardice is inexpensive insurance.

A minimal example looks like this:

```
DATA DIVISION.
FILE SECTION.
SD  SORT-WORK.
01  SORT-RECORD.
    05 SORT-NAME                PIC X(15) .

FD  NAMES-FILE.
01  NAMES-RECORD.
    05 NAMES-NAME                PIC X(15) .
```

```
FD  SORTED-FILE.  
01  SORTED-RECORD.  
   05  SORTED-NAME          PIC X(15).
```

[...]

```
    SORT SORT-WORK  
      ASCENDING KEY SORT-NAME  
      USING NAMES-FILE  
      GIVING SORTED-FILE.
```

Calling and Structuring Programs

The book-management application is split into two compiled modules. The main program provides the browse interface, while the second contains the add/edit functionality. The latter can also be run directly as a standalone Add application, while exposing a separate parameterised entry point when it is called to edit an existing record. Although calling another COBOL program is straightforward in principle, Realia has a few peculiarities here, particularly around program state and repeated calls within an ordinary linked DOS executable.

To distinguish the Edit path from the normal Add behaviour, I used the *ENTRY* statement. Realia supports multiple named entry points into the same compiled program correctly. The program's normal *PROGRAM-ID*, *BOOKADD*, is used in both cases where no parameter is required: when the program is run directly as a standalone utility, and when it is called from the main application to add a new book. Editing is different because the caller needs to tell the program which existing record is being edited, so a separate *ENTRY "BOOKEDIT" USING ...* provides that parameterised entry point.

This also avoids a subtler but real trap. A *LINKAGE SECTION* item describes storage supplied by a caller through *USING*. Running the program directly does not provide such an argument. Isolated tests with Realia COBOL 4.00 showed that a program declared with *PROCEDURE DIVISION USING* can nevertheless be run standalone without a runtime error. If the linkage item was left untouched, the program completed normally. Reading it returned leftover memory rather than an empty field, while writing to it also completed without any immediate error despite there being no valid caller-supplied value behind it. The failure mode is therefore potentially silent rather than a clean diagnostic. Using separate *BOOKADD* and *BOOKEDIT* entry points is not required by Realia, but is a safer design: the normal *BOOKADD* path never references linkage data, while *BOOKEDIT* does so only when it has actually been called with *USING*.

The more surprising behaviour appears when a called program is reused. With the ordinary multi-module executable produced by *LINK.EXE*, calling a subprogram, issuing *CANCEL*, and then calling it again does not work. The first call succeeds, but the second fails with:

```
ERCL0046: Recall after cancel invalid for EPDCALL old style .EXE
```

The practical rule is therefore simple: in this style of executable, do not *CANCEL* a subprogram that will be called again during the same run. Instead, have each entry point explicitly initialise whatever Working-Storage state needs to be reset before doing its work.

I also tested *PROGRAM-ID ... IS INITIAL PROGRAM*, since on paper it appears to offer exactly the behaviour required. An isolated subprogram containing a single Working-Storage counter was called repeatedly by a small driver with no explicit *CANCEL* anywhere. The first call behaved as expected, with freshly initialised data, but the second failed with the same ERCL0046 error. Realia recognises *IS INITIAL PROGRAM* as valid syntax, but with this executable structure it still leads to the same recall restriction encountered after an explicit *CANCEL*. Whatever mechanism the runtime uses internally, it does not provide a usable way to repeatedly reinitialise a called program in this kind of executable.

The *EPDCALL* compiler directive does not change that result either. I repeated the test with *EPDCALL* enabled in both caller and subprogram, as required by the later documentation, and confirmed through the compiler's own directive listing that it was active. An explicit *CALL*, *CANCEL*, *CALL* sequence still failed on the second call with the identical ERCL0046 message.

The wording of that message is significant: it refers to an “EPDCALL old style .EXE”. The later 4.2 documentation describes a different linking model using RealBIND and segmented multi-module executables or DLLs. I did not explore that build model as part of this project, so I would not treat it as tested here. What the experiments do establish is narrower and more useful: enabling *EPDCALL* alone does not transform an ordinarily linked executable into that architecture, nor does it make *CANCEL*-then-recall work.

For the kind of plain executable used by the book-management application, manual reinitialisation is therefore not merely an ugly workaround waiting for the right compiler switch. It is the reliable programming model: initialise the subprogram's state explicitly at the relevant entry point, call it as often as necessary, and leave *CANCEL* alone.

One part of the calling model is pleasantly uncomplicated. *GOBACK* adapts correctly to the way the program was entered. When the program has been called by another COBOL module, it returns control to the caller. When the same program is run directly, it terminates the process. Using *GOBACK* at the normal exit point therefore allows the same compiled program to serve both as a standalone utility and as a callable component without needing separate exit logic.

Dialect Limits and Surprises

Building the book-management application uncovered a handful of behaviours that did not fit neatly into the previous sections. Some were simply missing features. Others were cases where documentation from another Realia release suggested that something should work, only for the 4.00 compiler to have rather different ideas. The following are the ones that had a practical effect on the application rather than merely being historical curiosities.

Reference Modification Is Not Supported

I encountered one of the more significant limitations while implementing title search in the browse screen. Reference modification normally allows a program to operate on part of an alphanumeric field by supplying a starting position and an optional length, for example *FIELD(3:5)* to select five characters beginning at position three.

What made this surprising during the original investigation is that the later CA-Realia documentation available to me described reference modification plainly, including offset and length expressions. The DOS Realia COBOL 4.00 compiler tested here, however, rejects the syntax completely. The original Realia COBOL 3.00 manual obtained later does not appear to document the feature.

Any attempt to use the normal *identifier(start:length)* form was rejected beginning at the opening parenthesis. A typical sequence of diagnostics included:

```
Unrecognizable word or literal '('  
Expected data-name; found '1'
```

followed by further complaints about the colon and closing parenthesis. In other words, this was not a runtime peculiarity or a restriction on a particular expression: the parser simply did not recognise reference-modification syntax in the tested compiler.

This is a genuinely inconvenient omission for string processing. Ordinary table subscripting does work in the *PROCEDURE DIVISION*, but for the book search I used a different workaround based entirely on features already confirmed to behave correctly: fixed-length *REDEFINES* combined with *EVALUATE*.

The search string is stored once, then overlaid by a series of progressively longer views of the same storage:

```

01  WS-SEARCH-UPPER          PIC X(20) VALUE SPACES.

01  WS-SEARCH-V1
    REDEFINES WS-SEARCH-UPPER PIC X(1) .
01  WS-SEARCH-V2
    REDEFINES WS-SEARCH-UPPER PIC X(2) .
01  WS-SEARCH-V3
    REDEFINES WS-SEARCH-UPPER PIC X(3) .
01  WS-SEARCH-V4
    REDEFINES WS-SEARCH-UPPER PIC X(4) .

[...]
```

The computed search length then selects the matching view:

```

EVALUATE WS-SEARCH-LEN
  WHEN 1
    INSPECT WS-TITLE-UPPER
      TALLYING WS-MATCH-COUNT
      FOR ALL WS-SEARCH-V1
  WHEN 2
    INSPECT WS-TITLE-UPPER
      TALLYING WS-MATCH-COUNT
      FOR ALL WS-SEARCH-V2
  WHEN 3
    INSPECT WS-TITLE-UPPER
      TALLYING WS-MATCH-COUNT
      FOR ALL WS-SEARCH-V3
  [...]
END-EVALUATE
```

In the actual application I provided one *REDEFINES* entry for every supported search length, up to 20 characters. It is verbose, and reference modification would make the same operation considerably neater, but the technique is deliberately unambitious: *REDEFINES*, *EVALUATE* and *INSPECT* had all already been confirmed to work correctly on this compiler.

That made it a useful replacement for a missing language feature rather than another experiment balanced on top of the first one.

COMP-X Is Rejected

I came across *COMP-X* while investigating *CRT STATUS*. Before obtaining the closer Realia manuals, some of the material available to me described the three-byte status item using *COMP-X* for one of its fields. Testing that definition directly on Realia COBOL 4.00 showed that *COMP-X* is not accepted by this compiler.

Replacing it with *COMP-5*, however, worked correctly in the same *CRT STATUS* experiment. *COMP-5* is also used elsewhere in this project, notably for the numeric constants compared against *FIELD-TERMINATOR*, and is accepted by the compiler without difficulty.

The useful practical conclusion is therefore fairly simple: *COMP-X* was rejected in the tested Realia COBOL 4.00 environment, while *COMP-5* provided a working replacement in the case that originally led me to investigate it.

Report Writer

The Report Writer facility is not implemented in the Realia COBOL 4.00 compiler tested here. This was established experimentally before I had access to the original Realia manuals.

The compiler is unusually explicit about it. Attempts to use the Report Writer verbs produce diagnostics such as:

```
'INITIATE' Verb not supported by this version of the compiler
```

The same happens with *GENERATE* and *TERMINATE*. Introducing a *REPORT SECTION* produces additional syntax errors, but those are secondary to the more fundamental limitation: the Report Writer verbs themselves are not supported.

The original Realia COBOL 3.00 manual later provided useful corroboration. Words such as *REPORT*, *RD*, *GENERATE*, *INITIATE* and *TERMINATE* appear in its reserved-word list, but that list contains recognised reserved words rather than a catalogue of implemented features. More significantly, the manual's statement-format chapter explicitly omits unsupported facilities, and none of the Report Writer verbs appears there.

The Finished Application

The experiments described above eventually converged into a small but complete book-library application. The screenshots below show the resulting interface rather than introducing new techniques: the browse view, title search and add/edit form are all built from the SCREEN SECTION, FIELD-TERMINATOR handling and sequential-file techniques discussed earlier.

I have kept the captions brief because the mechanics behind each screen have already been covered in the preceding sections.

```
Book Library                                     Page:      1

Search:

Sel Title                                     Location
-----
1 Garage
3
4
4444 Home2
5
Another Test by the time Home
Foo Garage3
Foo Home
Locateme Home
Mastering COBOL Home
Principles of COBOL programming Garage
Programming in COBOL/400 Home

F3=Exit  F4=Search  F5=Top  F8=Next  F6=Add  E=Edit  D=Delete
```

Browse screen. The main application view, showing the current page of the sequential-file catalogue and the key-driven actions available from the list.

```
Book Library                                     Page:      1
Search: Cobol.....
Sel Title                                     Location
-----
Mastering COBOL                               Home
Principles of COBOL programming              Garage
Programming in COBOL/400                     Home

(blank = show all, Enter to apply)
F3=Exit  F4=Search  F5=Top  F6=Add  E=Edit  D=Delete
```

Search screen. Filtering the browse view by title substring. In the example, searching for “Cobol” returns the matching entries.

```
Book Library - Edit a book

ISBN      : 37737.....
Title     : Another Test by the time
Author    :
Year      : 0044
Location  : Home
Note      :

F3=Back  Enter=Next  Esc=Restore
```

Edit screen. The add/edit form, driven field by field through the *SCREEN SECTION* and function-key handling described earlier.

Appendix A: FIELD-TERMINATOR Reference

The full table below is taken from the Realia COBOL 4.2 User Guide, the closest surviving DOS documentation to the 4.00 compiler tested here.

During development I did not reconstruct the complete FIELD-TERMINATOR table. I used a small test program to display the value returned by FIELD-TERMINATOR, identified the keys the application actually needed, and then inferred the contiguous function-key values once the sequence became obvious. The book application therefore uses only a small subset of the available codes.

The later CA-Realia Workbench manuals document additional keyboard codes beyond those listed in the 4.2 guide. I have not included those later extensions in the table below.

Value	Key/Meaning
0	AUTO-ENTER signalled
26	End of file, when input is redirected
9	Tab
13	Return
15	Shift+Tab (Back Tab)
16–25	Alt+Q, W, E, R, T, Y, U, I, O, P
27	Esc
30–38	Alt+A, S, D, F, G, H, J, K, L
44–50	Alt+Z, X, C, V, B, N, M
59–68	F1 through F10
71	Home
72	Cursor Up
73	PgUp
75	Cursor Left
77	Cursor Right
79	End
80	Cursor Down
81	PgDn
82	Insert
83	Delete
84–93	Shift+F1 through F10
94–103	Ctrl+F1 through F10
104–113	Alt+F1 through F10

Value	Key/Meaning
114	Ctrl+PrtSc
115	Ctrl+Cursor Left
116	Ctrl+Cursor Right
117	Ctrl+End
118	Ctrl+PgDn
119	Ctrl+Home
120–131	Alt+1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, =
132	Ctrl+PgUp
133	F11
134	F12
135	Shift+F11
136	Shift+F12
137	Ctrl+F11
138	Ctrl+F12
139	Alt+F11
140	Alt+F12

The 4.2 documentation also confirms that Cursor Left and Cursor Right normally remain editing keys rather than terminating an *ACCEPT*. Enabling *ACCEPT-CURSOR-TERMINATORS* changes that behaviour, returning 75 for Left and 77 for Right.

For the book-management application itself, only a much smaller subset was required:

Value	Key/Meaning
9	Tab
13	Return
15	Shift+Tab
27	Esc
61	F3
62	F4
63	F5
64	F6
66	F8
71	Home
72	Cursor Up
80	Cursor Down

Those values are the ones actually exercised by the application rather than merely copied from documentation.

The same 4.2 chapter documents redirected input as well. When *ACCEPT* input comes from a file, keyboard echo is suppressed, CR terminates the field and LF is ignored. End of file produces *FIELD-TERMINATOR* = 26 once, after which subsequent input returns to the keyboard. Redirected *DISPLAY* output is written to the file including CR/LF sequences and does not receive a DOS Ctrl-Z byte unless one is explicitly displayed.

Appendix B: ACCEPT_CONTROL Reference

The following is based primarily on the *ACCEPT_CONTROL* interface documented in the Realia COBOL 4.2 User Guide, pages 8-26 through 8-28. The original Realia COBOL 3.00 manual documents an earlier form of the same interface, but the 4.2 manual is the closest surviving documentation to the Realia COBOL 4.00 compiler tested here and provides the fuller definition used below.

ACCEPT_CONTROL is a callable runtime subprogram that changes the behaviour of subsequent *ACCEPT* statements. The settings remain in effect until another call changes them or the program terminates. The routine is part of the standard runtime library and requires no special linking. The parameter block described by the 4.2 manual contains eight switches followed by six bytes reserved for future expansion:

```
01  ACCEPT-PARMS.
    05  ACCEPT-AUTO-ENTER          PIC X.
        88  SET-AUTO-ENTER        VALUE "Y".
        88  CLEAR-AUTO-ENTER      VALUE "N".

    05  ACCEPT-TEST-EMPTY         PIC X.
        88  SET-TEST-EMPTY        VALUE "Y".
        88  CLEAR-TEST-EMPTY      VALUE "N".

    05  ACCEPT-FILL-REQUIRED      PIC X.
        88  SET-FILL-REQUIRED     VALUE "Y".
        88  CLEAR-FILL-REQUIRED   VALUE "N".

    05  ACCEPT-CURSOR-TERMINATORS PIC X.
        88  SET-CURSOR-TERMINATORS VALUE "Y".
        88  CLEAR-CURSOR-TERMINATORS
                                VALUE "N".

    05  ACCEPT-ENTRY-REQUIRED     PIC X.
        88  SET-ENTRY-REQUIRED    VALUE "Y".
        88  CLEAR-ENTRY-REQUIRED  VALUE "N".

    05  ACCEPT-NO-ENTER-ECHO     PIC X.
        88  SET-NO-ENTER-ECHO     VALUE "Y".
        88  CLEAR-NO-ENTER-ECHO   VALUE "N".

    05  ACCEPT-MODE               PIC X.
        88  ACCEPT-IN-BIOS-MODE   VALUE "Y".
        88  ACCEPT-IN-DOS-MODE    VALUE "N".

    05  ACCEPT-SCREEN-MSG-OFF     PIC X.
        88  SET-SCREEN-MSG-OFF    VALUE "Y".
        88  CLEAR-SCREEN-MSG-OFF  VALUE "N".

    05  FILLER                    PIC X(6).
```

The routine is then called with:

```
CALL "ACCEPT_CONTROL" USING ACCEPT-PARMS
```

For each of the first eight bytes, “Y” enables the corresponding mode and “N” disables it. Other values leave the current setting unchanged. The final six bytes are reserved for future expansion, allowing the parameter block to be extended without changing its existing layout.

If *ACCEPT_CONTROL* is never called, all switches default off except *ACCEPT-SCREEN-MSG-OFF*, which defaults on. Settings established by a call remain active for later *ACCEPT* statements until another call changes them or the program terminates.

Parameter	Behaviour
<i>ACCEPT-AUTO-ENTER</i>	Applies to non-numeric fields. When a field becomes completely full, it is accepted automatically and <i>FIELD-TERMINATOR</i> is set to 0. It has no effect on numeric entry.
<i>ACCEPT-TEST-EMPTY</i>	If an empty field is terminated, the normal <i>FIELD-TERMINATOR</i> value is returned negated. For example, -71 indicates that Home was pressed while the field was empty.
<i>ACCEPT-FILL-REQUIRED</i>	Applies to non-numeric fields. The field must be completely filled before a terminator is accepted. An earlier terminator sounds the alarm and is ignored. It has no effect on numeric entry.
<i>ACCEPT-CURSOR-TERMINATORS</i>	Cursor Left and Cursor Right normally act as in-field editing keys. Enabling this option makes them terminators instead, returning 75 for Left and 77 for Right.
<i>ACCEPT-ENTRY-REQUIRED</i>	Applies to both numeric and non-numeric fields. An actual value must be entered. Attempting to terminate a null field sounds the alarm and the attempt is rejected.
<i>ACCEPT-NO-ENTER-ECHO</i>	Suppresses the CR/LF normally echoed when Enter terminates a field. This is useful when several fields are being accepted in sequence.
<i>ACCEPT-MODE</i>	“Y” causes keyboard input to be handled through the BIOS interface, allowing the scan code to be examined through <i>FIELD-TERMINATOR</i> . “N”, the default, uses DOS-level keyboard input.

Parameter	Behaviour
<i>ACCEPT-SCREEN-MSG-OFF</i>	“Y”, which is also the default when <i>ACCEPT_CONTROL</i> has never been called, disables <i>SCREEN SECTION</i> warning messages on the bottom line of the screen. Invalid input produces a beep instead. “N” allows the warning messages to be displayed.

The switches are particularly relevant when implementing field-by-field input. For example, *ACCEPT-CURSOR-TERMINATORS* allows Cursor Left and Cursor Right to participate in the application's own navigation logic rather than remaining invisible editing keys. *ACCEPT-AUTO-ENTER* provides a global automatic-entry mode, with an automatically completed field reported through *FIELD-TERMINATOR* = 0.

The book-management application did not ultimately require *ACCEPT_CONTROL*. It was investigated while working out Realia's keyboard-input model and is included here because it completes the picture around *ACCEPT*, *FIELD-TERMINATOR* and application-controlled navigation.

Appendix C: Indexed File Organization

This appendix sits slightly outside the rest of the project. The book-management application deliberately uses sequential files throughout, but while consulting the later Realia COBOL 4.2 manual I ran a small isolated test of indexed organization to see how the DOS 4.00 compiler behaved. None of this code was used by the book application itself.

An indexed file uses the familiar COBOL clauses:

```
SELECT IDX-FILE
  ASSIGN TO "IDXTEST.DAT"
  ORGANIZATION IS INDEXED
  ACCESS MODE IS DYNAMIC
  RECORD KEY IS IDX-KEY
  FILE STATUS IS WS-FILESTATUS.
```

No separate utility is required to create the file. A normal *OPEN OUTPUT* creates it, just as the later manual describes.

The test program created three records, attempted to write a duplicate primary key, reopened the file for *I-O*, performed a random read by key, traversed the file in key order with *START* and *READ NEXT*, and finally attempted to read a key that had never been written.

Operation	Result
Initial <i>OPEN OUTPUT</i>	File created successfully
Duplicate primary-key <i>WRITE</i>	<i>FILE STATUS 22</i>
Random <i>READ</i> by primary key	<i>FILE STATUS 00</i>
<i>START</i> followed by <i>READ NEXT</i>	Records returned in ascending key order
<i>READ</i> of a missing key	<i>FILE STATUS 23</i>

The primary key is therefore enforced as unique, with a duplicate *WRITE* returning the standard status 22 rather than damaging the file. Random access by moving a value into the record-key field and issuing a *READ* worked normally, while sequential access through *START* and *READ NEXT* returned records in key order. A lookup for a key that did not exist returned status 23.

The meanings of status 22 and 23 are also corroborated by GCMasterFile.COB, Eugenio Di Lorenzo's GnuCOBOL port of a 1992 CA-Realia indexed-file sample, which handles the same two values as duplicate key and record not found respectively.

For the operations tested here, indexed organization behaved as expected from conventional COBOL and was consistent with the later 4.2 documentation. The experiment was deliberately limited. I did not test alternate keys, *WITH DUPLICATES*, indexed *REWRITE* or *DELETE*, nor the Realia-specific block and buffer tuning parameters described in the later manual.

Indexed files nevertheless do not fit the design goal of the book-management application. Their on-disk representation is an implementation-specific binary format managed by the Realia runtime, rather than something intended to remain directly readable and transferable outside that environment. Sequential files are less convenient in several respects, but they remain inspectable, convertible and understandable without requiring the original runtime. For this project, that trade-off mattered more than indexed access.

So the indexed-file support is worth documenting because it works, but it remains an appendix rather than the road the application itself took.

Appendix D: An Implicit-Scope Nesting Trap

Realia COBOL 4.00 accepts traditional COBOL control flow without explicit scope terminators, but it also supports the ANSI85-style *END-IF*, *END-READ* and *END-PERFORM* forms. The original Realia COBOL 3.00 manual already documents explicit scope terminators as an ANSI85 feature, and direct tests confirm that they work correctly on the 4.00 compiler as well.

If the older implicit-scope style is used, however, it is easy to create a silent nesting error. Consider:

```
IF condition
    MOVE A TO OUT-RECORD
ELSE
    MOVE B TO OUT-RECORD
WRITE OUT-RECORD.
```

Without an *END-IF* or an earlier terminating period, the *WRITE* still belongs to the *ELSE* branch. It therefore executes only when the condition is false. The other branch falls through without writing anything, and the compiler has no reason to report an error.

The clearest fix on Realia 4.00 is simply to close the conditional explicitly:

```
IF condition
    MOVE A TO OUT-RECORD
ELSE
    MOVE B TO OUT-RECORD
END-IF

WRITE OUT-RECORD.
```

I confirmed *END-IF*, *END-READ* and *END-PERFORM* together in an isolated test containing a file read loop, nested conditional processing and no periods inside the block. The program compiled cleanly, read exactly the three records written by the test, and terminated correctly on *AT END*.

This is not a Realia-specific defect. It is simply a consequence of using implicit scope in older-style COBOL. Realia 4.00 gives you the much less surprising explicit alternative.

Appendix E: Sources and References

These notes are based on a mixture of direct testing, contemporary source code, surviving documentation and inspection of the compiler itself. Throughout the project, empirical behaviour on the actual Realia COBOL 4.00 compiler was treated as the primary source of truth. Documentation from other releases was used as supporting evidence and historical context rather than assumed to describe 4.00 exactly.

The most important evidence was therefore the compiler itself. Features and behaviours described in these notes were tested with small isolated programs wherever possible, particularly where documentation was absent, contradictory or belonged to another release. Several questions that remained ambiguous in the manuals were settled only this way.

Direct inspection of *REALCOB.EXE*, *REALCOB.LIB* and *REALDOS.LIB* also proved useful. Unix tools such as *strings*, together with symbol-table inspection, exposed internal routine names, compiler vocabulary and runtime diagnostic text that helped identify features worth testing experimentally.

Two contemporary CA-authored sample programs from 1992, obtained as *REALIA-SCRADDR.CBL* and *REALIA-ESCADDR.CBL*, were especially valuable when investigating screen handling and keyboard input. Their respective program IDs are *SCRADDR* and *ESCADDR*. Both implement name-and-address maintenance applications and use the same *FIELD-TERMINATOR* mechanism later confirmed experimentally. In several places these programs proved more useful than the surviving written documentation because they showed how CA itself expected the facilities to be used in real code.

The CA-Realia II Workbench documentation was another early source, including the 3.0 online COBOL Reference Guide and User Guide distributed as Windows Help files, and the later 3.2 documentation. The WinHelp files had to be extracted locally using a modified build of the open-source *helpdeco* utility, since I could not find a prebuilt version that worked satisfactorily on a modern system. These materials were useful as an initial map of the language and runtime, and were the source of several hypotheses tested during the project. They describe a substantially later product generation, however, so individual behaviours were treated cautiously until they could be confirmed against the DOS 4.00 compiler.

A much closer historical reference arrived considerably later: the original *Realia COBOL 3.00 User Manual*, published before the Computer Associates era. Its value is different from that of the Workbench material. It provides an earlier point in the product line,

making it possible to compare what already existed in Realia COBOL 3.00 with what was observed directly in 4.00 and what appeared later in CA-Realia releases.

This original Realia COBOL 3.00 manual should not be confused with the much later CA-Realia II Workbench 3.0 documentation despite the identical-looking version number. They belong to very different generations of the product. It is also more than historical background: as of this writing, Realia COBOL 3.00 and 4.00 are the two DOS compiler releases I have personally been able to locate in surviving software archives, making the 3.00 documentation directly useful to anyone experimenting with the DOS lineage today.

The closest surviving manual I was able to obtain to the compiler tested in this project is the *CA-Realia COBOL 4.2 User Guide for PC-DOS/OS2*, dated July 1992. This is a 644-page scan of a personal physical copy and was obtained only after the book-management application was already working. It belongs to the same DOS release family as the 4.00 compiler examined here and was far more directly relevant than the later Workbench documentation.

Its role was nevertheless mostly retrospective. Much of what it documented had already been discovered experimentally: *SCREEN SECTION* behaviour, *FIELD-TERMINATOR* and *ACCEPT* handling, program-call behaviour, the *EPDCALL* model and the restrictions around *CANCEL* and recalling subprograms in an ordinary linked executable. Where the 4.2 manual contributed information that had not already been established experimentally, that distinction is noted in the relevant sections of this document.

The indexed-file appendix also draws on *GCMASTERFILE.COB*, Eugenio Di Lorenzo's GnuCOBOL port of a 1992 CA-Realia indexed-file sample. It provides useful independent corroboration of two file-status values observed in the isolated indexed-file test: status 22 for a rejected duplicate key and status 23 when a requested record cannot be found. Those values were reached independently by the test program used for this project and by the older CA-authored application.

Access to both the original Realia COBOL 3.00 manual and the CA-Realia COBOL 4.2 manual was kindly provided by Eugenio Di Lorenzo, who is also responsible for the *GCMASTERFILE.COB* port mentioned above. His copy of the 4.2 manual still contains handwritten margin notes from an earlier reader working through some of the same material. Without those surviving physical manuals, this project would have had no contemporary documentation from the closest releases before and after the DOS 4.00 compiler being examined.

Appendix F: Example Programs

These example programs are included so that the document remains useful on its own. The repository is the canonical home for the source files, but if this PDF should survive separately from it, these four small programs provide enough working material to compile and link a program, use the SCREEN SECTION and FIELD-TERMINATOR, read and write a sequential file, and exercise Realia's native SORT.

They are deliberately small and purpose-built rather than complete applications. Their role is to provide a practical starting point alongside the notes above.

HELLO.COB: Minimal build

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
*
*   BSD Zero Clause License
*   Copyright (C) 2026 Tara Stella <tara@tara.sh>
*   See the included LICENSE file.
*
*   Minimal build example for CA-Realia COBOL 4.00 on DOS.
*   Two steps to build it:
*
*       REALCOB HELLO.COB,HELLO.OBJ,HELLO.LST
*       LINK HELLO.OBJ,HELLO.EXE,,REALDOS,,
*
*   REALCOB only compiles to an .OBJ file. Turning that into a
*   runnable .EXE is a separate step done by LINK.EXE, linked
*   against the REALDOS runtime library. See HELLO.BAT for both
*   steps together.
*
*   Source format: fixed-format COBOL. Columns 1-6 are the
*   sequence number area (unused here), column 7 is the
*   indicator area ('*' marks a comment line, like this one),
*   and actual code lives in columns 8-72.
*
PROCEDURE DIVISION.

0000-MAIN.
    DISPLAY "Hello from Realia COBOL 4.00, running under DOSBox".
    STOP RUN.
```

SCREEN.COB: Screen and keys

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. SCREEN.
```

```
*  
*   BSD Zero Clause License  
*   Copyright (C) 2026 Tara Stella <tara@tara.sh>  
*   See the included LICENSE file.  
*  
*   Minimal SCREEN SECTION example: two fields, entered one at  
*   a time, using FIELD-TERMINATOR to tell Enter apart from a  
*   couple of function keys. F1 quits without showing anything,  
*   F2 on the second field skips it (leaves it as zero).  
*
```

```
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
SOURCE-COMPUTER.  IBM-PC.  
OBJECT-COMPUTER.  IBM-PC.
```

```
DATA DIVISION.  
WORKING-STORAGE SECTION.
```

```
01  WS-NAME          PIC X(20).  
01  WS-AGE           PIC 99.  
01  WS-QUIT          PIC X          VALUE "N".
```

```
SCREEN SECTION.
```

```
01  ENTRY-SCREEN.  
    05 BLANK SCREEN.  
    05 LINE 2 COLUMN 2 VALUE "Name (F1=quit)      : ".  
    05 SCR-NAME LINE 2 COLUMN 26 PIC X(20) USING WS-NAME.  
    05 LINE 4 COLUMN 2 VALUE "Age (F2=skip)       : ".  
    05 SCR-AGE  LINE 4 COLUMN 26 PIC 99  USING WS-AGE.  
  
01  RESULT-SCREEN.  
    05 LINE 7 COLUMN 2 VALUE "You entered:".  
    05 LINE 8 COLUMN 2 PIC X(20) FROM WS-NAME.  
    05 LINE 9 COLUMN 2 PIC 99  FROM WS-AGE.
```

```
PROCEDURE DIVISION.
```

```
MAIN.
```

```
    DISPLAY ENTRY-SCREEN.
```

```
    PERFORM UNTIL WS-QUIT = "Y"
```

```
        ACCEPT SCR-NAME
```

```
        IF FIELD-TERMINATOR = 59
```

```
            DISPLAY SPACE WITH BLANK SCREEN
```

```
            DISPLAY "Cancelled with F1 - nothing else shown."
```

```
            MOVE "Y" TO WS-QUIT
```

```
        ELSE
```

```
            ACCEPT SCR-AGE
```

```
            IF FIELD-TERMINATOR = 60
```

```
        MOVE 0 TO WS-AGE
      END-IF
      DISPLAY RESULT-SCREEN
      MOVE "Y" TO WS-QUIT
    END-IF
  END-PERFORM.

STOP RUN.
```

SEQFILE.COB: Sequential files

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEQFILE.

*
*   BSD Zero Clause License
*   Copyright (C) 2026 Tara Stella <tara@tara.sh>
*   See the included LICENSE file.
*
*   Minimal SEQUENTIAL file example: writes three fixed-length
*   records, closes the file, reopens it, and reads them back.
*

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  IBM-PC.
OBJECT-COMPUTER.  IBM-PC.

INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SAMPLE-FILE
        ASSIGN TO "SEQFILE.DAT"
        ORGANIZATION IS SEQUENTIAL
        FILE STATUS IS WS-FILESTATUS.

DATA DIVISION.
FILE SECTION.
FD  SAMPLE-FILE.
01  SAMPLE-RECORD.
    05 REC-NAME          PIC X(15).
    05 REC-CITY          PIC X(15).

WORKING-STORAGE SECTION.
01  WS-FILESTATUS       PIC XX.
01  WS-EOF              PIC X          VALUE "N".

PROCEDURE DIVISION.

MAIN.
    OPEN OUTPUT SAMPLE-FILE.
    DISPLAY "Writing 3 records to sequential file SEQFILE.DAT".

    MOVE "Ada Lovelace"  TO REC-NAME.
    MOVE "London"        TO REC-CITY.
    WRITE SAMPLE-RECORD.

    MOVE "Grace Hopper"  TO REC-NAME.
    MOVE "New York"      TO REC-CITY.
    WRITE SAMPLE-RECORD.

    MOVE "Alan Turing"   TO REC-NAME.
    MOVE "Maida Vale"    TO REC-CITY.
    WRITE SAMPLE-RECORD.
```

```
CLOSE SAMPLE-FILE.

OPEN INPUT SAMPLE-FILE.
DISPLAY "Reading records".

PERFORM UNTIL WS-EOF = "Y"
    READ SAMPLE-FILE
    AT END
        MOVE "Y" TO WS-EOF
    NOT AT END
        DISPLAY REC-NAME " - " REC-CITY
    END-READ
END-PERFORM.
CLOSE SAMPLE-FILE.

STOP RUN.
```

SORTDEMO.COB: Native sort

IDENTIFICATION DIVISION.

PROGRAM-ID. SORTDEMO.

*

* BSD Zero Clause License

* Copyright (C) 2026 Tara Stella <tara@tara.sh>

* See the included LICENSE file.

*

* Minimal native SORT example: writes four names in random
* order to a file, sorts them with the USING/GIVING form (no
* INPUT or OUTPUT PROCEDURE needed), then reads the sorted
* result back. Called SORTDEMO rather than SORT since SORT
* itself is a reserved word and can't be a program name.

*

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. IBM-PC.

OBJECT-COMPUTER. IBM-PC.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT SORT-WORK

ASSIGN TO "SORT.TMP".

SELECT NAMES-FILE

ASSIGN TO "NAMES.DAT"

ORGANIZATION IS SEQUENTIAL.

SELECT SORTED-FILE

ASSIGN TO "SORTED.DAT"

ORGANIZATION IS SEQUENTIAL.

DATA DIVISION.

FILE SECTION.

SD SORT-WORK.

01 SORT-RECORD.

05 SORT-NAME PIC X(15).

FD NAMES-FILE.

01 NAMES-RECORD.

05 NAMES-NAME PIC X(15).

FD SORTED-FILE.

01 SORTED-RECORD.

05 SORTED-NAME PIC X(15).

WORKING-STORAGE SECTION.

01 WS-EOF PIC X VALUE "N".

PROCEDURE DIVISION.

```

MAIN.
  OPEN OUTPUT NAMES-FILE.
  MOVE "Turing"    TO NAMES-NAME.
  WRITE NAMES-RECORD.
  MOVE "Lovelace" TO NAMES-NAME.
  WRITE NAMES-RECORD.
  MOVE "Hopper"   TO NAMES-NAME.
  WRITE NAMES-RECORD.
  MOVE "Babbage"  TO NAMES-NAME.
  WRITE NAMES-RECORD.
  CLOSE NAMES-FILE.

  SORT SORT-WORK
    ASCENDING KEY SORT-NAME
    USING NAMES-FILE
    GIVING SORTED-FILE.

  OPEN INPUT SORTED-FILE.
  PERFORM UNTIL WS-EOF = "Y"
    READ SORTED-FILE
      AT END
        MOVE "Y" TO WS-EOF
      NOT AT END
        DISPLAY SORTED-NAME
    END-READ
  END-PERFORM.
  CLOSE SORTED-FILE.

*   CA-Realia has its own DELETE FILE statement (not standard
*   ANSI COBOL) for removing a closed file from disk - handy
*   for cleaning up work files like NAMES.DAT once you're done
*   with them. Left commented out here so this demo's files
*   stay behind for you to look at:
*
*       DELETE FILE NAMES-FILE.

  STOP RUN.

```

LICENSE for the source files

BSD Zero Clause License

Copyright (C) 2026 Tara Stella <tara@tara.sh>

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.