Cobolito/400: A Tiny Data Appliance in the Lineage of the IBM 5280

A tribute to forgotten small systems, clear contracts, and data-first machines.

Tara Stella

Essay written by:
Tara Stella
July 2026

E-mail: tara@tara.sh
Web site: https://tara.sh

To my tech partner,
wherever you are.

Note to the Reader

This essay is a research and design tribute to the IBM 5280 Distributed Data System and to a wider lineage of small business machines, record-oriented files, removable media, and explicit operational contracts.

It follows the path that led to Cobolito/400: punched cards, diskettes, the IBM 3740, the IBM 5280, IBM midrange systems, Distributed Data Management, TULIP/400, and finally a small modern data appliance built from an Alpine Linux kernel, an initramfs, COBOL line sequential files, and a proof-of-life application called CrumbStation.

Cobolito/400 is tiny and experimental. It is not a product announcement, not a retrocomputing reenactment, and not an attempt to make the 1980s happen again, which is probably just as well. I built it because I wanted to understand, in a practical way, the shape of a small data appliance: one clear task, local records, removable media, operator actions, and data that can leave the machine cleanly.

In that sense, Cobolito/400 is not the main subject trying to steal the stage. The main subject is the design lineage that made systems like the IBM 5280 possible. Cobolito/400 is my small experiment after following that trail.

Where possible, I have relied on manuals, IBM papers, product literature, archive material, books consulted, implementation notes, and personal recollection. Wikipedia was useful mostly as a map: a way to find names, dates, adjacent systems, terminology, and paths for further digging.

Any mistakes, misunderstandings, or overenthusiastic crumbs made while researching, experimenting, or writing this essay are mine.

Table of Contents

The Idea Behind Cobolito/400

Cobolito/400 began as a way to understand a kind of computing I kept returning to: small systems with clear jobs, explicit data contracts, local operation, and a visible path for moving data elsewhere.

The practical question was simple: could that shape still make sense if translated into modern materials?

I did not want only to describe an older style of data system. I wanted to build a small working instrument that could test the idea: boot into one task, collect records locally, and provide a clean way to move those records out. Not a clone, not a full reconstruction, and not a commercial product, but a proof of life.

This essay is therefore part historical reading, part technical reconstruction, part system-design exercise, and part personal tribute.

Much of it came from following manuals, IBM Systems Journal papers, product announcements, archive fragments, and conversations with people who were closer to that world than I ever was. The result is not nostalgia for beige plastic. It is an attempt to understand a way of designing business systems where records, files, operators, media, and procedures all had clear roles.

Project Quiet Ground, later becoming TULIP/400 as an implementation, grew from the same research. Some parts exist. Some parts are still design notes. But the contracts were never an afterthought. They were the point: libraries, file formats, data movement, local operation, and clear boundaries between one small system and another.

Cobolito/400 is one small tribute inside that larger path. It exists because the IBM 5280 showed me a shape of computing I wanted to preserve: a small system, one clear job, local records, explicit media, and understandable operation.

The following chapters describe how I found that shape, why the IBM 5280 mattered to me, and how Cobolito/400 became my small modern experiment after following that trail.

Finding the IBM 5280 while Researching Project Quiet Ground

Over the years, I noticed that the systems I felt most at home in shared a specific trait. It took me a long time to see the bigger picture and name it clearly. I have always been attracted to complex engineering, networking, and large distributed architectures, but part of me was also drawn to smaller, more tangible, self-contained systems.

Eventually I understood that I was looking for a low-noise environment: a place where the machine did one thing clearly, where the data had a visible shape, and where the operator was not constantly pulled through chats, interruptions, notifications, and accidental complexity.

So I started digging into the history of the software and hardware I liked: IBM i and midrange systems, BSD and Unix, lost business systems, and my beloved dBase III+. I wanted to understand why certain technological choices had been made, and what trade-offs shaped them.

I did not have the language for it yet, but I was looking for data-centric computing.

I was not simply searching for something old. I was searching for an operating value that mattered to me: the belief that data should outlive programs, that copying should remain meaningful, that recovery should be boring, and that systems should respect both resources and the people who run them.

Because I could not find something that fitted me, and because I could not simply buy a modern IBM i machine for personal experimentation, I decided to design my own small midrange-like environment. Not for fame, and not to recreate the past, but to create a focused, single-task space that made sense to me. Project Quiet Ground became the codename for that research and design path. TULIP/400, named after my flower and as a tribute to the AS/400, became the implementation.

During late 2025 and early 2026, I spent a great deal of time following what I came to think of as the lineage of Project Quiet Ground. I read manuals, IBM papers, old product material, and archive fragments. I followed the trail through IBM business systems, especially the System/3x world, System/38, and AS/400. I also looked at VMS Record Management Services, HP 3000 TurboIMAGE, PickOS, MAPPER, dBase, and other systems where data was not treated as an afterthought.

Much of the design documentation for these systems is hidden, scattered, or no longer public. Many manuals still exist only because of preservation work done by archives,

collectors, and people who decided that old documentation was worth saving. During that research, I stumbled upon the IBM 5280.

This was not a weekend of retrocomputing browsing. It became a small act of design archaeology: manuals, architecture papers, product announcements, old chronology pages, and conversations with people who had lived closer to that world.

The 5280 immediately caught my attention. It was not a general-purpose machine. It was a system made with purpose: local data entry, local processing, removable media, optional communication, and a clear operational workflow. It felt like a small machine carrying some of the same design instincts I admired in larger midrange systems. Data and contracts were not an afterthought. They were part of the system's DNA.

That was the moment Cobolito/400 began to make sense.

I did not want to clone the IBM 5280. I wanted to pay tribute to it, and to the forgotten business machines that were never famous, but quietly carried the daily work of many companies. Cobolito/400 became my way of preserving part of that idea in a smaller, modern, personal form.

From Punched Cards to Diskettes: The Road to the IBM 5280

Punched cards were used both to program computers and to collect business information. Operators could capture data locally according to a defined layout, then send the cards back to a central facility where the expensive systems lived: initially electromechanical tabulating machines, later mainframes such as the IBM System/360 and System/370, or midrange systems such as the IBM System/3 family.

At that time, the structure of the punched card itself was the contract. It was physical. The card layout had to match the expectations of the central system, whether that meant an electromechanical control panel or a later computer program capable of interpreting the fields.

In 1973, IBM introduced the 3740 Data Entry System as a successor to traditional punched-card data entry for centralised, decentralised, and remote environments. It was a strange and fascinating class of machine: a desk-like data-entry station with a keyboard, one or more 8-inch diskette drives depending on the model, and a small display.

The key element was the IBM diskette. IBM described it as a reusable magnetic medium that replaced the standard punched card. One diskette could hold 1,898 records, roughly the equivalent of one box of 80-column cards. Unlike punched cards, the diskette could be corrected, updated, reused, and searched by machine.

The IBM 3740 translated the punched-card workflow into digital form. Records were still entered locally, much like on a keypunch, but they were written to diskette after the complete record had been keyed. The diskette could then be physically sent to another department or to a converter, or, in remote configurations, the data could be transmitted through binary synchronous communications.

That independence was itself a contract: data captured in one place, in a known format, so that it could be processed somewhere else. The IBM 3740 did not yet define Distributed Data Management, but it pointed toward the same architectural instinct. Before the network contract, there was already a media contract.

The IBM 5280 would take this idea further. It kept the small, dedicated, data-entry shape, but moved closer to the programmable, distributed, business-machine lineage that connects the 3740 world to the System/3x and later IBM midrange systems.

The IBM 5280: How a Small Data-Entry System Worked

In 1980, IBM announced the IBM 5280 Distributed Data System. It followed the IBM 3740 in the lineage of machines that had replaced punched-card data entry with diskette-based local capture, but it was not merely a better keypunch. It was a small programmable system built around diskettes, local processing, and optional communication with larger IBM systems.

The IBM manuals describe the 5280 as a multipurpose, diskette-based system for data entry, remote job entry, and data processing. A 5280 installation could consist of a programmable data station or programmable control unit, auxiliary workstations, diskette drives, printers, and communications capability.

In modern terms, one might be tempted to call it an edge system, but that word risks hiding what made it interesting. The 5280 was not just a remote terminal. It was a local data machine with its own programs, data sets, storage, displays, printers, and operational procedures.

The 5280 inherited part of the IBM 3740 world. It still understood the importance of local data capture and diskette exchange, and IBM provided migration paths from 3740 Application Control Language programs to the 5280 assembler environment. At the same time, the 5280 was more ambitious. It had a real software stack behind it, not just a data-entry keyboard: programming languages, operational utilities, and a path for COBOL applications.

This is one of the reasons the 5280 feels closer to the IBM midrange world than to the personal computers of the same period. It belonged to a different design tradition: a business data system, built around records, operators, procedures, and local work rather than a general-purpose desktop.

A Family of Small Data Machines

The 5280 was not a single box. It was a family.

The 5285 Programmable Data Station was a single-operator tabletop system. The 5286 Dual Programmable Data Station was a dual tabletop programmable keyboard/display station with two keyboards and two diskette drives. This made it a curious and memorable machine: two operators could sit at the same unit, facing a shared piece of equipment, each working at a keyboard/display station.

The 5288 Programmable Control Unit provided a larger clustered configuration, with auxiliary data stations, more diskette drives, printers, and communications capability.

This modularity mattered. The 5280 could be used as a small single-user data-entry station, a dual-operator station, or part of a larger distributed data-entry installation. Depending on the configuration, it could support local data capture, printing, sorting, merging, communication with a host, and more than one active program.

The system's storage was divided into partitions. Each program executed in a partition, and the system could execute more than one program at a time. Foreground partitions were associated with interactive keyboard/display use; background partitions could be used for work that did not require continuous operator interaction, such as utilities or communications tasks. This made the 5280 feel less like a passive terminal and more like a small operating environment.

Programs, Utilities, and the Working Day

The 5280 system was built around a set of IBM program products.

System Control Programming (SCP) provided the basic system functions: IPL, configuration, recovery, and patching. DE/RPG provided a high-level programming environment for data-entry and business applications. Assembler was available for lower-level or migrated applications, especially for users coming from the IBM 3740 ACL world.

The Utilities Program Product provided common operational functions, such as allocating and deleting data sets, entering data, converting 3740 programs, copying diskettes, and displaying system status. Sort/Merge provided a way to sort records in a data set or merge records from two data sets. Communications utilities supported data transfer to and from other systems when the hardware configuration included communications capability.

COBOL support was different from DE/RPG and assembler. The COBOL program was not normally compiled on the 5280 itself. IBM provided host compilers for OS/VS and DOS/VSE, allowing programmers to write 5280 COBOL applications on the host side, compile them there, and then move the resulting program to the 5280. That movement could happen by diskette or through a communications link, depending on the installation.

This is an important distinction. The 5280 was programmable, but not in the same way as a later personal computer with a self-contained development environment for everything. It belonged to an IBM business ecosystem. Programs, diskettes, host systems, communications lines, printers, and operators were all part of the overall workflow.

IPL and Loading a Program

From the operator's point of view, the 5280 had a particular rhythm.

At startup, the system performed an Initial Program Load (IPL). The SCP diskette loaded a starter system configuration. The system then carried out internal checks and prepared the environment needed to load and run other programs. A user-defined IPL diskette could also be created through the system configuration program, so that a particular installation could tailor the system to its own storage, devices, and daily work.

Once the system was ready, the operator could load a program. The 5280 load prompt asked for the information needed to run it: the program name, the device address, and the partition number where the program should execute.

That small prompt says a great deal about the machine.

A program was not an icon in a desktop environment. It was not a cloud service. It was a named object loaded from a device into a partition. The operator had to know, or be guided to know, where the program lived and where it should run. The machine made the operating contract explicit.

This is one of the reasons the 5280 feels so different from later general-purpose computers. The system did not try to hide its operational model completely. Programs, devices, partitions, and diskettes were part of the user's visible world.

Data Sets as the Centre of the System

The 5280 was built around data sets. Data was not an afterthought hidden inside an application. Data sets were allocated, copied, deleted, exchanged, sorted, merged, recovered, and processed by system utilities and application programs.

The access methods also show the record-oriented nature of the machine. The 5280 supported sequential access, direct access by relative position, and key-indexed access. These ideas were familiar in business programming environments such as COBOL, RPG, VSAM, and IBM midrange systems. They also connect the 5280 to the later discussion of Distributed Data Management, where record-oriented files become one of the core architectural concerns.

This is where the 5280 becomes especially relevant to the story of Cobolito/400. The important idea is not only that data existed on diskette. The important idea is that the data had a defined shape and operational meaning. A diskette was not just removable storage. It was part of a workflow: capture data locally, preserve it as a data set, copy it, exchange it, transmit it, or process it elsewhere.

The 5280 manuals also discuss data exchange with other IBM systems and compatibility paths with the IBM 3740 under specific diskette formats. This continues the older punched-card and 3740 pattern: local data is collected in one place, then consumed in another place through an agreed format.

Screens, Records, and COBOL

For my own research, the COBOL support was especially interesting because it exposed two areas where vendor-specific systems often reveal their real character: file handling and screen handling.

COBOL as a language has long been standardised, but real COBOL applications have always depended on the environment around the compiler. File assignment, device handling, indexed files, screen interaction, printers, and terminals are where the host system enters the language.

On the IBM 5280, COBOL programs could assign files to diskette devices through the FILE-CONTROL paragraph. In a SELECT entry, the ASSIGN TO DISK clause reflected the physical and operational reality of the machine: data lived on named media and devices. A program could specify the diskette device, volume identifier, and data set name, or leave enough unspecified that the operator would be prompted. File sharing was also part of the file-control vocabulary, allowing another program to use a file at the same time when permitted.

Screen handling also had an IBM flavour. The 5280 documentation used Data Description Specifications for defining data-entry formats. These were not merely cosmetic screens. They described records, fields, checks, prompts, and display behaviour. This belongs to the same broad IBM habit that appears elsewhere in the midrange world: screens and records are described through structured definitions, not improvised one character at a time by every application.

This is one of the bridges between the 5280 and later systems such as the System/38 and AS/400. I do not mean that the 5280 was an AS/400 ancestor in a simple straight line. The point is more subtle: the same design vocabulary appears again and again in IBM business systems. Records matter. Fields matter. Files matter. Displays are part of the data contract. Operators interact with described workflows.

The Quiet Lesson of the 5280

The IBM 5280 did not become one of the famous machines. It is not remembered like the IBM PC, the System/360, or the AS/400. Even when I worked around AS/400 support and later spent time in Rochester, I do not remember encountering one directly.

That may be part of why it interests me. The 5280 sits in a quiet corner of computing history. It was specialised, diskette-based, programmable, operationally explicit, and designed for real business data. It could live close to the people entering the data, while still fitting into a larger IBM processing environment.

The 5280 was not a personal computer. It was not a mainframe terminal. It was not merely a keypunch replacement. It was a small data system with a job to do.

That is the part I wanted to remember.

Records, Screens, and Business Machines: The IBM Midrange Vocabulary

The IBM 5280 should not be described as a direct ancestor of the AS/400. That would be too simple, and probably wrong. What matters here is not a straight genealogical line, but a shared design vocabulary.

I do not have a document proving that the IBM 5280, System/38, System/36, and later AS/400 were designed as one single architectural programme. But the overlap is too strong to dismiss as coincidence. The 5280 was developed in Rochester, in the same General Systems Division environment that was also responsible for the System/3x lineage and, later, the AS/400. It used a vocabulary immediately recognisable from IBM business systems: RPG, COBOL, assembler, records, data sets, utilities, screens described through structured definitions, printers, operators, device names, and command-like interfaces.

This does not make the 5280 part of a simple family tree. It makes it part of the same design culture, a shared dialect spoken inside the same engineering house.

In that world, the centre of gravity is not the program as an isolated object. It is the business record: fields, files, data sets, screens, printers, operators, and procedures arranged around the movement and validation of data.

This is why the 5280 feels familiar when viewed from the midrange side. It was programmable with languages such as DE/RPG and assembler, and COBOL support existed through host compilers. Its documentation talks in terms of data sets, records, access methods, utilities, device addresses, partitions, and operator workflows. These are not the words of a general-purpose personal computer. They are the words of a business data machine.

The screen model is part of the same pattern. In the 5280 documentation, Data Description Specifications were used to describe data-entry formats. The important point is not that these were identical to the DDS display files later associated with System/38 and AS/400, but that they reflect the same habit: screens are described as structured fields and records, not merely drawn as free-form pixels. The display is part of the data contract.

That idea appears again, more strongly, in the AS/400 and IBM i world. Files, display files, printer files, jobs, authorities, and database definitions are part of the operating

environment. Applications do not invent the whole world from scratch. They live inside a system that already understands business data.

This is one of the ideas that Project Quiet Ground and TULIP/400 try to preserve in a much smaller form. Cobolito/400 is not an AS/400, and it is not a 5280 clone. But it borrows from this vocabulary: records first, screens as data-entry instruments, explicit files, operator actions, and clear paths for moving data elsewhere.

IBM Distributed Data Management: Contracts as Architecture

Contracts have always been the basis of how computers exchange data, whether through physical media, files, or a network. A contract is a promise between whoever produces data and whoever consumes it: a common, stable way to exchange information.

That may sound like an old idea, but it is still central today. When people describe REST APIs with OpenAPI or Swagger specifications, or define contracts between microservices, they are dealing with the same basic problem: one side produces information, another side consumes it, and both need a shared understanding of structure, meaning, and behaviour.

The idea did not begin with networks. In punched-card workflows, the card layout itself was the contract. Local operators recorded data in a physical structure that central systems were prepared to interpret. The IBM 3740 moved part of that contract into digital form: data could be captured locally on diskette and later consumed by another IBM system.

IBM Distributed Data Management, or DDM, formalised this instinct further. Instead of relying only on a shared medium or file format, it defined architectural agreements for creating, managing, and accessing data across systems.

DDM was formalised in IBM architecture manuals, but today the material is scattered and unevenly accessible. It is often encountered through later product documentation, implementations, and secondary references rather than through one easily available narrative source. One useful reference would be the IBM Distributed Data Management Architecture: General Information manual, GC21-9527-02, although it is now difficult to find.

According to the available literature, DDM was IBM's published software architecture for managing distributed data. It was initially designed to support record-oriented files. Later, it was extended to support hierarchical directories, stream-oriented files, queues, and system command processing. It also became part of the foundation for IBM's Distributed Relational Database Architecture, DRDA, and was later extended around data description and conversion.

DDM sits close to IBM's broader distributed-systems world, including SNA/APPC and the Rochester midrange context. But what caught my attention was not the communications mechanism itself. The important point, for this essay, is DDM level 1: record-oriented

files. DDM recognised that a file contract was needed before exchange could be made reliable across systems.

The System/36 file system had been defined to meet the record-oriented needs of third-generation programming languages such as Fortran, COBOL, PL/I, and IBM RPG. So had the System/38 file system and the VSAM record-oriented data sets used on IBM mainframe systems. Yet their actual facilities and interfaces varied considerably. A common contract was needed between systems, and DDM defined one around record-oriented files.

DDM defines four kinds of record-oriented files: sequential files, direct files, keyed files, and alternate index files.

Sequential files store records in consecutive slots. Direct files store records in numbered slots, allowing records to be accessed sequentially, relative to an access-method cursor, or randomly by slot number. In the usual COBOL vocabulary, this corresponds to relative file organization. Keyed files store records in consecutive slots while maintaining a secondary order through an index of key-field values. Alternate index files provide a separate index of key-field values based on an existing sequential, direct, or keyed file.

For those familiar with COBOL, these categories are not surprising. They are also not unique to IBM. VMS Record Management Services, or RMS, has a similar record-oriented spirit. Even DBF files from the dBase III+ era belong to a neighbouring world: structured records in files that can be opened, copied, inspected, and processed by more than one program.

This mattered to me because Project Quiet Ground, the codename for what became TULIP/400, was file-based by intention. If I wanted systems to exchange data, even locally and offline, I had to define the contracts first. The same was true for Cobolito/400, my small tribute to the IBM 5280. Data would not be exchanged through a network at first, but through removable media: USB disks instead of floppies. The principle was still the same.

A file is not just bytes. A record is not just text. A diskette, a USB stick, or a network protocol only becomes useful when both sides understand what is being exchanged.

That is the lesson I took from DDM: distribution begins with a contract.

Project Quiet Ground (TULIP/400) Data Contracts

There were many experiences that influenced Project Quiet Ground. I described some of these choices in my blog post “Data First, Programs as Guests,” where I first tried to explain the idea that data should remain the centre of gravity and that programs should orbit around it rather than trap it.[1] Three influences became especially important when I started thinking about data contracts.

The first was dBase III+. dBase was not just a database program. It was an environment. DBF files were the centre of gravity: screens, reports, and bits of code orbited around them. You could copy a floppy disk and carry the whole small world with you: data, structure, and logic together.

What mattered to me was that the data format was inspectable, documented, and portable. The structure was not hidden entirely inside the program. The data could outlive one particular implementation. Long before SQLite and embedded databases became common, you could move a useful system by copying files. It felt like having your world on a floppy disk. You carried it with you, and it was all there.

The second influence was the AS/400, now IBM i, which I encountered directly while working for IBM, both in Europe and during my time in Rochester. IBM i takes data seriously at the operating-system level. The database is not merely an external component installed on top of the system; it is integrated into the system itself. Files are first-class objects. Data structure is defined and enforced by the environment, not improvised separately by every application. In that world, the contract is not only a file format. The contract is part of the system.

The third influence came from Linux and from one of the encounters that shaped my professional life: Carlo Strozzi, creator of the original NoSQL database. The Wikipedia entry[2] is useful as a public pointer to that project and its place in database history, but my connection to it was also personal: I knew Carlo, and I saw some of that work from close range. Although I was working in the AS/400 world, I already had a strong interest in Linux. Through Carlo’s work, and through testing and small contributions around it, I saw how simple and powerful plain-text data files could be. They were not glamorous, but they were understandable, portable, and easy to process with ordinary tools.

1 https://www.tara.sh/posts/2026/2026-02-16_data_first/
2 https://en.wikipedia.org/wiki/Strozzi_NoSQL

These experiences pointed in the same direction. In the systems I leaned toward, data was the centre of gravity, not programs. When data is primary, copying data is already meaningful. Backup, restore, migration, and disaster recovery do not always need to become complex rituals. If data is well structured and its contract is clear, copying it is preservation. Programs can be redeployed, rewritten, or replaced, but the meaning can remain.

This became a fundamental concept in TULIP/400: the core file formats had to be explicit. Before building applications, I needed to decide what kinds of data contracts the environment would recognise.

The initial contracts are CSV, COBOL line sequential files, SQLite, and DBF.

CSV is mostly for personal use and simple data. It is not glamorous, but it is durable, obvious, and easy to process.

COBOL line sequential files are for data that should remain open, transferable, and compatible across systems. Indexed COBOL files are useful, but their format often depends on the compiler implementation and linked runtime library. Line sequential files are less clever, but easier to carry across boundaries.

SQLite is for more structured data where I want an open, single-file database rather than compiler-specific indexed files.

DBF belongs partly to affection, but not only to affection. It represents the dBase tradition: simple records, structure carried with the data, and a file that can be copied, inspected, and understood outside one program.

These formats are also the planned backends for the Data File Utility (DFU) I am developing for TULIP/400. The goal is not to build a full application framework, but something closer in spirit to the original IBM i DFU tool (STRDFU) and dBase: simple, direct interaction with data, without unnecessary ceremony.

A Smaller Machine, the Same Vocabulary

Earlier, I described the relationship between the IBM 5280 and the later IBM midrange world as a matter of shared vocabulary rather than proven descent.

The relationship between TULIP/400 and Cobolito/400 is different.

Cobolito/400 is not a mode of TULIP/400, and it is not simply TULIP/400 made smaller. It is a separate appliance project, closer in spirit to the IBM 5280. But its design ideas are intentionally borrowed from TULIP/400. The inheritance is deliberate.

TULIP/400 is the fuller system. It assumes a more capable local environment, multiple libraries, and several data contracts. Cobolito/400 takes only the part that makes sense for a lighter data-entry appliance. For the first version, that means the data contracts that travel most easily: CSV and COBOL line sequential files.

The same narrowing happens with libraries. In TULIP/400, libraries are distinct places where related programs, data, and definitions can live together. Cobolito/400 reduces that idea almost to its smallest useful form. There is only the data area: the application, its runtime pieces, its supervisor script, and its data files all live together under one small roof.

So the pattern repeats at a smaller scale, but this time deliberately. IBM Rochester built systems such as the IBM 5280 and AS/400 from its own business-computing vocabulary. I am not pretending to stand at that scale. Cobolito/400 is a tiny personal appliance built on a desk, not inside one of the great engineering houses.

But I learned from the best engineers I could find: the ones who left their ideas in manuals, systems, file formats, operator workflows, and machines that still have something to teach.

Different scale, different materials, similar choice: put the data first, make the contract visible, and let the machine have a clear job.

Cobolito/400: A Tiny Data Appliance, Not a Clone

What interested me in the IBM 5280 was not nostalgia for a specific machine, but a way of thinking: a small dedicated system, close to where data is created, booting into a clear task, storing records locally, and providing an explicit way to move those records elsewhere.

I loved that idea.

I wanted a small system that would preserve the same pattern in modern form: start quickly, ideally with an almost instant-on effect, enter one specific data task, collect records, and provide a clear way to copy them out.

Cobolito/400 is my tribute to that shape of computing. It is not an IBM 5280 clone. It does not emulate the hardware, reproduce the operating system, or attempt to rebuild the exact environment IBM shipped in 1980. That was never the point.

The point was translation.

The 5280 did this with diskettes, IBM data sets, partitions, utilities, operator prompts, and optional communications. Cobolito/400 does it with a Linux kernel, a small initramfs, COBOL line sequential files, an application supervisor, and FAT USB export.

The materials are different. The contract is similar.

Cobolito/400 starts from a simple idea: the appliance should have one job, and the user should not have to pass through a general-purpose operating system to reach it. Power on the machine, load the small environment, enter the application, collect the data, copy it out, and shut it down. No desktop, no package manager in the operator path, no hidden database server, no network dependency, and even a hard power-off should not turn the system into a drama. Just a small data machine with a visible workflow.

Choosing the Base

My first instinct was FreeDOS.

That was not accidental. FreeDOS has the kind of immediate feeling I wanted: a quick boot, a direct path into one program, and an operating style where the machine does not pretend to be more complicated than it is. It also has the old “turn it off when you are done” feeling, provided the application has written its data safely and no filesystem operation is in progress.

For one old machine, that might have been enough. But Cobolito/400 was not meant to depend on a single target. I wanted to sideload it from an existing Linux laptop, boot it from USB, and eventually run it on small dedicated hardware. That made the boot environment part of the design problem. FreeDOS is happiest in a legacy BIOS or CSM-style world, while many modern systems are UEFI-only or have poor legacy support. I briefly looked at compatibility paths such as CSMWrap, but that felt like a clever workaround rather than a reliable appliance foundation. The base of Cobolito/400 should not depend on persuading a modern machine to pretend too hard that it is an older one.

I also considered FreeBSD. A small memory-resident FreeBSD appliance is entirely possible, and projects such as mfsBSD show that the idea is sound. But for this implementation, FreeBSD would have meant spending much more time learning a different appliance-building path.

Linux was familiar ground. I had worked for commercial Linux distributions, including Red Hat, Canonical/Ubuntu, and SUSE, and part of that professional life involved building, customising, integrating, debugging, and maintaining Linux-based systems. That experience also taught me a useful warning: every local deviation from upstream becomes something you own.

Linux From Scratch was tempting for the same reason it was dangerous. It would have made Cobolito/400 feel very self-contained, but it would also have turned the project into a private distribution. The first successful build is not the real cost. The real cost is every rebuild, every security update, every dependency change, and every future day when the old build process has quietly rotted.

So I chose Alpine Linux as the base, but not as a normal installed operating system. Cobolito/400 uses a stock Alpine LTS kernel and a small initramfs assembled from Alpine components. That gave me a maintained foundation while keeping the operator path small.

This was a maintenance decision as much as a technical one. Cobolito/400 is intentionally small, but I did not want it to become fragile. A tiny appliance should be understandable, but it should also remain something I can put down and pick back up later without rediscovering an entire private operating system.

The Appliance Contract

Before building too much, I had to decide what Cobolito/400 actually was.

Was it only one application, or was it an appliance layer that could host small data applications?

I decided it should be an appliance.

That meant defining a boundary between the underlying environment and the application space. TULIP/400 already had the idea of libraries: self-contained places where programs and data can live together in a coherent unit. Cobolito/400 borrows a much smaller version of that idea. In the appliance, the data area is the application space. It contains the application program, its data, and the related runtime pieces needed by that application.

The appliance layer is responsible for preparing the machine quickly: boot the kernel, unpack the initramfs, mount the basic runtime filesystems, load the required modules, set up the terminal, and hand control to the application supervisor.

The application layer is responsible for the actual workload.

In the first proof-of-life system, that workload is CrumbStation. But CrumbStation is not inseparable from Cobolito/400. It can also run outside the appliance, for example on a normal Linux system, on FreeBSD, or in another small environment such as my PursePC project. When it runs inside Cobolito/400, the supervisor provides an appliance context through environment variables and return codes. That lets the same application gain appliance-specific behaviours, such as Copy/Save and power handling, without hardwiring those behaviours directly into the COBOL program.

From the operator's point of view, this should still feel like one coherent system. Internally, however, the boundary matters. Cobolito/400 prepares the small machine; the application performs the work; the supervisor translates application requests into appliance actions.

That boundary is the contract.

Small Enough to Be an Appliance

I did not want an entire operating system in the operator path. I needed a kernel and enough userland to avoid reinventing everything, but mostly I needed something able to run a COBOL screen application and copy data in and out through removable media.

Ideally, the appliance would run from RAM only, almost as if the operating environment were firmware. My target scenarios were simple: sideload the appliance from an existing Linux laptop, boot it from USB, or eventually run it on a dedicated device that had the tactile feeling of a small embedded terminal.

For that physical target, I chose an old WeTab tablet. It is an x86 machine with 1 GB of RAM and a 16 GB internal SSD, modest enough to make the constraints real, but still

modern enough to boot a Linux kernel. It also has the right physical feeling for this project: closer to a small point-of-service or data-entry terminal than to a general-purpose laptop.

The WeTab was useful because it was constrained without becoming an immediate architecture port. Starting with x86 meant I could move from laptop testing to physical hardware while keeping the first build focused on the appliance shape itself.

I also wanted to use a USB barcode scanner. Cobolito/400 does not strictly require one, but the scanner made the idea of local data capture tangible. A scanner, a small screen, a keyboard, a local record, and a removable device: that was the physical vocabulary I wanted the system to have.

The hardware work was not glamorous, but it mattered. A data appliance that cannot reliably see a keyboard, a barcode scanner, or a USB stick is not an appliance. It is a small philosophical object with no hands.

Booting Into Purpose

Because the target was speed and simplicity, I wanted to reduce boot time far below the fifteen seconds I had achieved with PursePC, my earlier small Alpine-based mini-laptop project.

The kernel load takes some time, but much of the delay in a normal Linux system is not the kernel itself. It is the general-purpose startup path: service discovery, background daemons, hardware probing, logging, network preparation, and the many small ceremonies that make sense for a normal computer but not for a tiny data appliance.

So I wrote a small init script.

Calling it an “init system” would be far too grand. It mounts the basic runtime filesystems, loads the selected modules, prepares the environment, and starts either the application supervisor or a fallback shell. The work was not in inventing a new operating system, but in deciding how little was actually needed.

One practical problem was terminal handling. Dropping into a shell from init is easy. Running ncurses applications, such as nano and later COBOL screen programs, requires the process to have a proper controlling terminal. A normal Linux system would usually arrange that through getty or a similar mechanism. I did not want to add that machinery just to run one appliance program, so I used cttyhack, a small utility that provides the missing controlling-terminal step.

That was one of those tiny discoveries that feels larger than it looks. A few kilobytes of the right tool can remove an entire layer of unnecessary ceremony.

The result is narrow, but that is the point. Cobolito/400 does not wake up and ask what kind of computer it should be today. It wakes up already knowing its job.

Removable Media and Export

A data appliance is only useful if the collected records can leave the machine cleanly.

In the IBM 5280 world, removable media meant diskettes. In Cobolito/400, the practical equivalent is a small FAT-formatted USB stick. I did not want a full Unix mounting workflow, with the operator dealing with paths, mount points, filesystem state, and manual cleanup. I wanted something closer to an appliance action: insert media, choose the target, copy the data out.

That led me back to mtools, an old but still useful toolset for working with FAT filesystems without mounting them. FAT and VFAT remain widely readable, and mtools can operate directly on a FAT block device. That made it possible to create a simple Copy/Save ritual without exposing the operator to the machinery underneath.

The export utility belongs to the appliance layer rather than to one specific application. Different applications may generate different records, but the basic export action is part of the appliance: take the local data files and copy them to removable media in a form another system can read.

That is why CrumbStation does not perform the export itself. It asks for an export through the supervisor contract. The appliance layer then runs the Copy/Save utility and returns to the operator flow.

This keeps the shape clear. The application records events. The appliance moves data.

The Data Contract

The final contract was the data itself.

Any application hosted under Cobolito/400 has to declare a data contract. That is part of what makes it an appliance application rather than simply a program that happens to run inside the initramfs.

TULIP/400 can support several file contracts because it assumes a fuller local environment. Cobolito/400 narrows that idea. It is a small data-capture appliance, not a general-purpose data system, so its first contract is deliberately limited to the formats that leave the machine most cleanly: CSV and COBOL line sequential files.

That narrowing is not a judgement against SQLite or DBF. Both remain useful in the wider TULIP/400 world. They simply ask more of the receiving side: a driver, a library, a binary file format, or a more capable local environment. Cobolito/400 starts from a stricter rule. The file should remain meaningful after it leaves the appliance.

The trade-off is familiar by now. CSV and line sequential files make some operations less convenient, especially browsing backwards, counting pages, or deleting individual records. There is no index quietly doing that work. In exchange, the data can be inspected, copied, archived, and imported without dragging the original application behind it.

CrumbStation is the first application to live under that policy. Its specific file contract is described in the next chapter, and the full record layout and import experiments are preserved in Appendix H.

Copying should be enough to begin the next step.

Appliance Proof of Life

The first working result was intentionally small: a stock Alpine LTS kernel, a self-contained initramfs, a custom init script, selected hardware modules, terminal handling, an application supervisor, USB export, and enough GnuCOBOL runtime to run CrumbStation.

On a modern laptop, the appliance can be sideloaded and reach the application environment in about one second. On the old WeTab tablet, from pressing the physical power button to reaching the application environment, it takes about five seconds.

USB keyboard input works. The barcode scanner works. USB storage works. Nano works as an ncurses test case. CrumbStation runs as a COBOL screen application. The operator can enter records, browse them, request Copy/Save, and power the appliance down through the supervisor flow.

The implementation trail is preserved in the appendices: base-system choice, initramfs build, boot sequence, hardware notes, terminal handling, removable-media export, supervisor contract, data format, and runtime choices. Those details matter, but the main result is the shape.

At that point, Cobolito/400 was no longer only a design note.

It had become a tiny data appliance.

CrumbStation: The Proof-of-Life Application

The name Cobolito/400 gives part of the story away. I wanted a COBOL application.

Part of that came from the tribute itself. COBOL belongs naturally to the IBM 5280, midrange, and business-data world I had been studying. But there was also a practical reason: compiled COBOL programs can be surprisingly small, especially when the runtime environment is already present. And, simply, I wanted to understand COBOL better. Not because I wanted to become a COBOL developer, but because curiosity kept pulling on the cable.

The question was what kind of demonstration application would make sense. It had to be small enough to build, but not so artificial that it failed to show the point of Cobolito/400. I wanted something that carried the spirit of local data collection: punched cards, the IBM 3740, the IBM 5280, and a modern removable-media contract.

The inspiration came from small local grocery shops in villages in the UK. Many of those shops are tiny universes. They sell food, but they may also act as the local post office, parcel counter, message board, and informal centre of gravity. So I imagined a small shop terminal that could log simple events: earning or spending loyalty points, scanning parcels, and recording local transactions.

That terminal became CrumbStation.

The name “Crumbs” honours the British tradition of tea and biscuits, but the application itself has a serious purpose. It is the first proof that Cobolito/400 can boot into a task, collect structured records, and export them without needing a full desktop, database server, or network connection.

The File Contract

Under the Cobolito/400 data contract, CrumbStation writes its records to a COBOL line sequential file. In this first version, the file is CRUMBS.DAT.

CSV is also allowed by the appliance policy, but it was not the natural choice for CrumbStation. This is a COBOL application, and line sequential files belong naturally to the language. GnuCOBOL can write one fixed-layout record per line without asking the application to become a CSV parser, with all the small ceremonies of quoting, escaping, delimiters, and edge cases arriving at the counter.

For a proof-of-life application, that mattered. I wanted CrumbStation to prove the appliance shape: collect records locally, write them clearly, and make them easy to move

elsewhere. Spending the first version on CSV handling would have proved a different point.

The practical payoff appears on the receiving side. A COBOL line sequential file is a good fit for a record-oriented import path, especially into IBM i once a matching Physical File exists and the transfer handles line endings correctly. That is not magic, and it is not automatic, but it is a clean contract: one line, one logical record, one known layout.

This does not mean every receiving system understands the file by instinct. A future back-office flow still needs to know the record layout, character set, line endings, and event meanings. But that is a manageable import problem, not a trapped-data problem.

That mattered to me.

The file is not the application. It is the handoff.

The Operator Screen

The main screen of CrumbStation is the part the operator would use most. It has to collect data quickly, because in the imagined setting there may be a customer standing in front of the counter. Scan the loyalty card or parcel, choose the function, write the record, done.

The screen also acts as the operator's main front-end to the appliance. From there, the operator can request a USB copy, browse the recorded data, exit back to the supervisor, or ask for the appliance to power off or reboot.

I decided to use function keys because I love that style of interface, and because it belongs naturally to the IBM i and midrange world: visible actions, clear keys, and 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.

The function keys are also part of the contract between the COBOL application and the appliance supervisor. CrumbStation does not itself copy files to USB or power off the machine. Instead, it exits with specific return codes. The supervisor interprets those return codes and performs the requested appliance-level action.

This keeps the boundary clean. CrumbStation handles the shop task. Cobolito/400 handles the appliance task.

Browsing and Deleting Records

While developing CrumbStation, I realised that I needed a way to inspect the collected data and delete records when necessary. That became the browse/delete function.

This is not meant to be the main operating mode of the application. It is a service screen: a way to check what is there, correct mistakes, and remove records before export if needed.

It also made the trade-off of the file contract visible. A line sequential file is easy to copy and import, but it is not a tiny database table in a false moustache. The program reads through the file in order. Moving forward is natural. Moving backward, counting pages in advance, or deleting selected records requires more explicit application logic.

That was acceptable for CrumbStation. The demo application is intentionally small, and the file contract matters more than providing every convenience of a larger database. The more detailed COBOL lessons from the browse/delete work are kept in the implementation notes.

Station and Device Identity

A further question appeared once records were being written: how should each transaction be identified?

Date and time, even down to hundredths of a second, were useful but not enough by themselves. I wanted the record to say where it came from and which appliance produced it. So I added a station name and a device name.

This serves two purposes. It helps make records unique when data from more than one appliance is imported later, and it preserves operational meaning. A transaction is not only something that happened. It happened at a station, on a device.

Because the appliance runs mostly as a read-only environment, I wanted configuration to be simple and external. The solution was to pass configuration through the Linux kernel command line. A bootloader entry can define the station and device values, the supervisor reads them, and the COBOL application receives them through the environment.

This is not meant to be a polished product mechanism. It is a proof-of-concept contract: simple, visible, and easy to change per physical appliance.

End-to-End Proof of Life

CrumbStation proved what I needed it to prove.

Cobolito/400 could boot quickly into a COBOL application. The application could collect records through a simple operator screen. It could store them in a copyable line sequential file. It could request appliance-level actions through return codes. It could identify the station and device that produced the data. And it could hand data back to the outside world through the Cobolito/400 USB export path.

That made CrumbStation more than a toy demo. It was the first small workload that exercised the whole idea: local capture, clear records, explicit operator actions, removable media, and data that could leave the appliance cleanly.

A tiny shop counter, perhaps. But enough to prove the machine had a pulse.

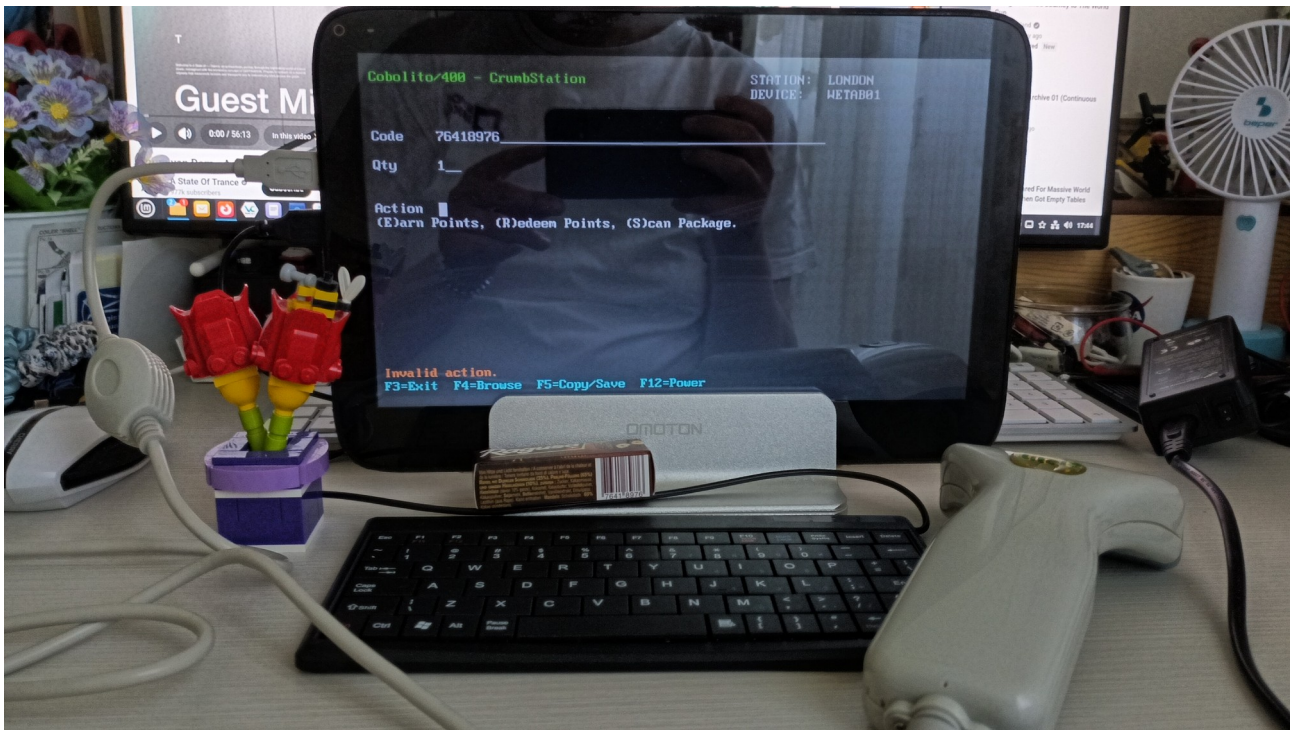

Figure 1: Cobolito/400 running as a physical CrumbStation appliance.
The WeTab is shown on a small metal stand in a compact desk setup, running the CrumbStation main screen with a USB keyboard, barcode scanner, and a scanned chocolate-bar code in front of it. The two tulips on the desk are a small nod to TULIP/400, the larger project whose data-contract ideas Cobolito/400 deliberately narrows into appliance form.

What Small Systems with Clear Contracts Still Teach Us

I loved every part of this journey: researching the IBM 5280, tracing the lineage backwards, reading manuals and papers, and, most of all, trying to understand why certain choices were made.

I keep saying that this was not nostalgia. Or perhaps more honestly: it was not nostalgia for the systems or programs themselves. It was nostalgia for a kind of system design where things were thought through coherently, where a machine was designed with its surrounding world in mind, and where a path forward to other systems was part of the plan.

I remember being in the IBM Rochester Labs in 1999, when people were already preparing for the idea of a common hardware direction across AIX, AS/400, and Linux, something that would become visible years later in the Power Systems world. That kind of planning stayed with me. It was not perfect, and improvisation certainly existed in the past too, but it did not feel like the default strategy. Today, too much technology feels like "put it in production and see what happens."

This journey reminded me that small systems can still teach serious design lessons.

Contracts matter. Local operation matters. Understandability matters. Copying data out cleanly matters. Operators matter. Doing one thing well matters.

I learned more about how to design a system with clear contracts. I learned more about COBOL, not only as a language, but as a way of thinking about records, files, screens, and efficient business applications. I learned from manuals, from architecture papers, and from people who were there when some of these ideas were being built. They have my deepest admiration.

I do not have reliable sales data or a precise end-of-life date for the IBM 5280, but it seems to have appeared at an awkward historical moment. It was a specialised distributed data system arriving just as general-purpose personal computers were becoming more flexible, cheaper, and culturally dominant. That may help explain why it is not widely remembered today, even though it represented an important class of business machine.

That is partly why I wanted to write this, and why I wanted to pay tribute with Cobolito/400. The 5280 may not have become famous, but the ideas around it still matter. Or, at least, they mattered to me.

I know Cobolito/400 is small. It is not an IBM system, not a commercial product, not a revolution, and not an attempt to drag the world back to 1980.

But small systems can still carry serious ideas.

A machine can boot into one clear task. A record can have a contract. A file can be copied without dragging an entire stack behind it. A system can be understandable enough that the operator is not treated as a nuisance. These were not primitive ideas. They were disciplined ideas.

Cobolito/400 is my way of saying that I noticed.

And perhaps, somewhere between an IBM 5280 manual, a COBOL line sequential file, a USB stick, and a tiny data-entry screen, there are still a few useful crumbs left on the table.

Appendices

The main essay explains the design path and the reasons behind Cobolito/400. These appendices collect the supporting implementation notes: build choices, boot details, hardware findings, terminal handling, removable-media export, application contracts, data layout, runtime decisions, COBOL lessons, and screenshots.

They are not meant to be a second essay. They are the juicy bit for the geeks. They are closer to field notes: the practical details I wanted to preserve after the machine finally worked, including a few wrong turns, useful commands, small discoveries, and the bits of wiring that made the proof of life possible.

Appendix A: Base System and Maintenance Contract

The main essay explains why Cobolito/400 ended up using Alpine Linux rather than FreeDOS, FreeBSD, Linux From Scratch, or a fully custom base. This appendix keeps the implementation side of that decision.

Cobolito/400 is small, but it is not built from nothing. That was deliberate. The goal was not to create a private operating system, a custom Linux distribution, or a museum of local patches that would work beautifully once and then quietly rot in a corner. The goal was to build a tiny appliance using maintained components wherever possible, and to customise only the integration layer.

The final base is Alpine Linux, used in a narrow way. Cobolito/400 uses Alpine's published miniroot filesystem material, a stock Alpine LTS kernel, selected userland tools, and only the modules needed by the first appliance targets.

The stock kernel choice may look conservative, especially for a project that cares about boot time. A custom kernel could probably remove unused features and save a little time. But a custom kernel also becomes a maintenance contract. The cost is not the first successful build. The cost is every future rebuild, every security update, every hardware variation, and every moment when an old local configuration no longer matches the world around it.

For Cobolito/400, I preferred a few extra milliseconds to a private kernel I would eventually stop updating.

The project therefore owns the appliance integration, not the whole operating system. Alpine provides the maintained base. Cobolito/400 assembles the initramfs, selects the needed pieces, provides the custom init script, adds the application area, and defines the appliance workflow.

That is the maintenance contract:

Cobolito/400 is small not because everything was rewritten, but because most things were deliberately not rewritten.

Appendix B: Initramfs Build Process

Cobolito/400 is built as a kernel/initramfs pair.

The build script does not create a full disk image. There is no installed root partition, no operator-facing package manager, and no normal booted Alpine system. Instead, the script prepares a small root filesystem tree, packs that tree into a compressed initramfs, and copies the Alpine LTS kernel beside it.

In the tested build, the output directory contained two files:

```
-rw-r--r-- 1 tara tara 5.6M Jun 16 17:24 cobolito
-rw-r--r-- 1 tara tara  14M Jun 16 17:24 vmlinuz-lts
```

The cobolito file is the initramfs. It contains the small userland environment, the custom init script, the selected appliance utilities, the required kernel modules, and the application payload. In the first proof-of-life build, that payload includes CrumbStation.

The vmlinuz-lts file is the Alpine LTS kernel used to boot the appliance.

Build Inputs

The script derives the Alpine version and machine architecture from the build host:

```
ALPINE_FULLVER="$(cat /etc/alpine-release)"
ALPINE_MAJORVER="${ALPINE_FULLVER%.*}"
MACHINE="$(uname -m)"
ALPINE_ROOTFS="alpine-minirootfs-$ALPINE_FULLVER-$MACHINE.tar.gz"
```

From those values, it knows which Alpine miniroot filesystem archive to use.

The working directories are created relative to the script location:

```
.cache/
  rootfs/
  output/
```

The rootfs directory is the working root filesystem tree.

The output directory receives the generated boot files.

Cached Root Filesystem

The build process supports both fast iteration and clean rebuilds.

By default, the script reuses the existing root filesystem tree if it is already present. This makes normal development faster, because the Alpine minirootfs does not need to be downloaded, extracted, and repopulated every time.

When the script is run with the clean option, or when the root filesystem tree does not exist, it creates a fresh root filesystem.

During a fresh build, the script downloads the Alpine minirootfs only if the archive is not already present in the cache. If the archive is already cached, it reuses it.

```
if [ "$1" = "clean" ]; then
    REBUILD_ROOTFS=1
fi
```

This gives the build a useful rhythm: fast rebuilds while changing appliance or application files, and a clean path when the base system needs to be regenerated.

Package Selection

After extracting the minirootfs, the script temporarily copies the host resolver configuration into the new root filesystem so that package installation can work inside the chroot.

It then installs a deliberately small package set:

```
ncurses
mtools
gmp
nano
```

Those packages are included for practical reasons.

ncurses provides the terminal library path needed by screen-oriented tools and COBOL screen programs.

mtools provides the removable-media Copy/Save mechanism used for FAT USB export.

gmp is required by the GnuCOBOL runtime used in the appliance.

nano is not part of the final operator workflow, but it was useful during development as a small ncurses test case inside the initramfs.

Kernel and Module Material

Cobolito/400 uses the Alpine LTS kernel from the build host:

```
/boot/vmlinuz-lts
```

The script copies that kernel into the output directory at the end of the build.

The initramfs also needs selected kernel modules. Instead of copying the whole module tree, the script copies only the modules needed by the first hardware targets and appliance workflow.

The selected module list covers USB core support, USB storage, USB controller drivers, SCSI disk handling, HID input, and the Linux input event layer:

```
usbcore
uas
usb-storage
ehci-hcd
xhci-hcd
ohci-hcd
uhci-hcd
ohci_pci
xhci_pci
ehci_pci
sd_mod
scsi_mod
usbhid
hid
hid_generic
evdev
```

For each selected module, the script uses modprobe to ask for the dependency list, then copies the required module files into the root filesystem while preserving their paths:

```
modprobe --show-depends $module | awk '{print $2}' | xargs -I{} cp --parents {}
$ROOT
```

After the module files are copied, the script copies the module metadata files and runs depmod inside the root filesystem.

This means the kernel, module directory, and module metadata are expected to belong together. They all come from the same Alpine host environment.

Appliance Files

Once the base root filesystem exists, the script adds the Cobolito/400-specific files.

The custom init script is installed as:

```
/init
```

The terminal helper is installed as:

```
/bin/cttyhack
```

The Copy/Save utility is installed as:

```
/bin/cobolito-copy
```

The application area is recreated on each build:

```
/data
```

The script removes any existing data directory from the working root filesystem, creates it again, and copies the contents of the local data directory into it.

This is useful during development. The base root filesystem can be reused from cache, while the application payload is refreshed each time the build runs.

In the first Cobolito/400 build, the data area contains CrumbStation, its related files, the GnuCOBOL runtime material needed by the application, and the supervisor script used to start the appliance workflow.

Directory Layout

The generated initramfs contains a small Linux filesystem tree. The most important paths are:

```
/init
/bin
/data
/dev
/proc
/sys
/lib/modules
```

The init script is the first userspace program run by the kernel.

The bin directory contains small appliance-level tools.

The data directory is the application space.

The dev, proc, and sys directories are mount points for runtime filesystems.

The lib/modules directory contains the selected kernel modules and dependency metadata.

Cleanup

Before creating the final initramfs, the script removes temporary or host-specific material from the working root filesystem:

```
/root/.ash_history
/etc/resolv.conf
/var/cache/apk/*
```

The goal is to avoid carrying local shell history, host resolver configuration, or package cache material into the appliance image.

Creating the Initramfs

The initramfs is created by entering the working root filesystem, packing it with cpio in newc format, and compressing it with gzip:

```
find . | cpio -H newc -o | gzip > $TARGET/cobolito
```

The Alpine LTS kernel is then copied beside it:

```
cp /boot/vmlinuz-lts $TARGET/
```

At that point, the output directory contains the two boot artifacts:

```
.cache/output/cobolito
.cache/output/vmlinuz-lts
```

A bootloader can then load vmlinuz-lts as the kernel and cobolito as the initramfs.

Build Contract

The build script follows the same design principle as the rest of Cobolito/400: reuse maintained components, keep the local integration explicit, and avoid turning the project into a private Linux distribution.

Alpine provides the maintained base. The build script assembles the appliance image. The init script starts the runtime environment. The data area contains the application world.

That is the build contract: assemble a tiny appliance image without pretending to own the whole operating system.

Appendix C: Boot Sequence and Custom Init

Cobolito/400 does not start a normal Alpine Linux userspace.

In a regular Alpine system, the kernel would eventually hand control to the standard init system, usually OpenRC, which would then run the normal service startup sequence. That is useful for a general-purpose machine, but Cobolito/400 is not trying to become a general-purpose machine. It is trying to become an appliance.

The boot sequence is deliberately smaller:

```
bootloader
  ↓
Linux kernel
  ↓
initramfs unpacked into memory
  ↓
/init
  ↓
basic runtime filesystems mounted
  ↓
required modules loaded
  ↓
environment prepared
  ↓
/data/cobolito-start
```

This is not a new operating system. It is Linux being used in one of its simplest forms. The kernel needs a first userspace program to run. On a normal installed system, that program is usually a full init system. In an initramfs, it can simply be a custom init script. In emergency or experimental situations, Linux can be started with something as small as init=/bin/sh.

That simplicity is easy to forget today, because most machines hide it behind service management, device discovery, logging, dependency ordering, networking, and distribution policy. Cobolito/400 deliberately avoids most of that machinery.

Starting Point

The initramfs contains a small custom init script. Once the kernel has unpacked the initramfs and started it, that script is responsible for preparing the appliance environment.

The first job is to mount the basic runtime filesystems:

```
mount -t devtmpfs devtmpfs /dev
mount -t proc proc /proc
mount -t sysfs sysfs /sys
```

These are enough to give the small environment access to device nodes, process information, and kernel/sysfs state.

Loading the Required Modules

After mounting the core filesystems, the init script loads the modules needed by the first Cobolito/400 hardware targets and appliance workflow.

In broad terms, this prepares the appliance for USB keyboards, USB storage devices, and the USB barcode scanner. The detailed hardware and module-selection notes belong in Appendix D. This appendix only records where module loading fits in the boot sequence.

Preparing the Environment

After loading the modules, the init script prepares a small operator and application environment.

It sets the terminal type:

```
export TERM=linux
```

It disables shell history:

```
export HISTFILE=/dev/null
export HISTSIZE=0
```

This is mostly cosmetic and hygienic. Cobolito/400 is not meant to preserve an operator shell history inside the appliance image. It also sets the home directory:

```
export HOME=/data
```

This reflects the appliance/application boundary. The data area is where the application world lives: the supervisor, the COBOL program, data files, and related application material.

Finally, the init script clears the screen and prints a small banner:

```
Welcome to COBOLITO/400!
```

That banner is not technically important, but it is operationally useful. It marks the point where the machine has stopped being a generic booting kernel and has become Cobolito/400.

Starting the Appliance Supervisor

The current boot contract is that the init script looks for an executable supervisor at /data/cobolito-start. If it exists, the init script starts it through cttyhack:

```
if [ -x /data/cobolito-start ]; then
    exec /bin/cttyhack /data/cobolito-start
fi
```

This was not the first version of the appliance. During early development, the goal was simply to boot into a usable shell, then make enough of the environment work to run tools such as nano. That was useful because it tested whether the initramfs had the right libraries, packages, terminal behaviour, and device support.

Only later did /data/cobolito-start become the explicit supervisor contract between the appliance layer and the application layer.

The distinction matters. The init script does not know the application logic. It prepares the machine and hands control to the supervisor. The supervisor then decides how to start CrumbStation, interpret return codes, copy data, open a developer shell, or request shutdown and reboot actions. Those details belong in Appendix G.

Fallback Shell

If /data/cobolito-start is not present or not executable, the init script falls back to a shell:

```
exec /bin/cttyhack /bin/sh
```

This fallback was important during development. A broken or missing application should not make the appliance image unusable. The system can still boot into a minimal shell for inspection and repair.

It also preserves the original experimental rhythm of the build: first prove that the tiny environment can boot, then prove that it can run terminal tools, then prove that it can run the application.

Terminal Handoff

Both the supervisor and the fallback shell are started through cttyhack.

That is not accidental. Running a shell from init is easy, but running interactive terminal programs correctly requires a usable controlling terminal. This mattered for nano, ncurses testing, and later COBOL screen programs.

The controlling-terminal issue is described separately in Appendix E. Here it is enough to note that cttyhack is part of the boot handoff. The init script does not merely execute the supervisor; it executes it in a way that gives the interactive application a usable terminal.

The Narrow Boot Path

Cobolito/400 boots quickly because the boot path is narrow.

There is no service discovery phase, no network preparation, no desktop session, no package manager, and no general-purpose startup choreography. The kernel loads, the initramfs is unpacked, the minimal runtime filesystems are mounted, the required modules are loaded, the environment is prepared, and control passes to the appliance supervisor.

That narrowness is not only about speed. It is part of the appliance contract. The boot path does not prepare a general-purpose computer and then ask the user what to do with it. It prepares one small environment for one known workload.

Appendix D: Hardware and Module Notes

Cobolito/400 was not meant to remain only a virtual experiment.

I wanted physical hardware. Not a laptop pretending to be an appliance, and not just a virtual machine window, but hardware I could touch: a small unit with a keyboard, a display, and the physical feeling of a data-entry station. Cobolito/400 was inspired by machines that lived close to the work being done, so I wanted the proof-of-life system to have at least some of that physical presence.

Reality, however, had opinions.

At the time, I was travelling often for work, and carrying separate hardware was not practical. I also had limited space at home. I had an external monitor for work, but not much room for another permanent test bench. So the first development path had to be portable: start in a virtual machine, then sideload the appliance kernel and initramfs from my normal laptop while experimenting wherever I happened to be, including hotel rooms.

That gave me a practical workflow. I could build and test the appliance image on modern hardware, while still keeping the goal of running it later on a dedicated physical device.

Development Targets

Target	Role in the project	Relevant characteristics	Result
VirtualBox VM	Build and controlled test environment	Used to build the appliance image, build the trimmed GnuCOBOL runtime, build CrumbStation, and test the workflow in a repeatable environment	Provided a consistent build and runtime base, including the specific library dependencies later carried into the appliance
Laptop sideload	Portable real-machine testing	Existing Linux installation, GRUB-based sideload path, available while travelling	Booted Cobolito/400 rapidly and reached the application environment in about one second
WeTab tablet	First dedicated physical appliance target	x86 tablet, 1 GB RAM, 16 GB internal SSD, USB input, small terminal-like form factor	Booted directly into Cobolito/400 and reached the application environment in about five seconds

This sequence reflects how the project actually developed. The virtual machine came first because it was practical, but also because it gave me a controlled build environment. The same VM built the appliance image, the GnuCOBOL runtime used by the appliance, and CrumbStation itself, so the resulting binaries and library dependencies belonged to the same small world.

The laptop sideload path made the appliance portable during development. The WeTab came later, when Cobolito/400 needed a body.

I also considered reusing a Raspberry Pi Zero I already had in a box, and in some ways that might have felt even closer to the original idea of a small terminal. But for the first working version, I did not want to turn Cobolito/400 into a cross-architecture porting exercise before the appliance shape itself had been proven.

The WeTab is not powerful, elegant, or modern in any useful marketing sense, which made it a good target. It has the physical feeling of something closer to a small point-of-service or data-entry terminal than to a general-purpose laptop. Cobolito/400 did not need glamour. It needed a machine that could sit there, accept input, save records, and get on with the job.

Input and Removable Media

The important accessories were a USB keyboard, a USB barcode scanner, and a FAT-formatted USB stick.

They are not the main hardware targets, but they are essential to the appliance ritual. The keyboard provides ordinary operator input. The barcode scanner behaves like keyboard input and gives CrumbStation a practical data-capture device. The FAT USB stick provides the removable export medium used by the Copy/Save workflow.

The exact kernel module list and copying mechanism are described in Appendix B. From the hardware point of view, the lesson was simpler: the appliance had to see USB input and USB storage early enough to be useful.

One detail mattered more than expected: older USB controller support.

Without UHCI support, the WeTab could boot far enough to look alive, but USB keyboard input did not work properly where I needed it. Adding the older USB controller support fixed that part of the puzzle.

That is the kind of detail that feels ridiculous until it is the one thing between a working appliance and a very small sculpture.

Internal Storage

The first Cobolito/400 build did not rely on the WeTab internal SSD for application data.

That was intentional at the appliance-design level, but also practical. The early image focused on USB input and removable export rather than on discovering every internal storage device. SATA/libata support was not part of the first selected module set, and the internal SSD was therefore not used by the appliance environment.

This may change later. A future build could add internal-storage support, or use the SSD as a local persistent area. For the first version, however, I preferred to keep the storage contract narrow: boot the appliance, write local records in the application area, and copy data out through removable media.

Bootloader Detour on the WeTab

The WeTab also produced a bootloader detour.

The original idea was to treat it like the development laptop: install or reuse a small Linux boot setup, add a Cobolito/400 boot entry, and sideload the appliance through a normal bootloader configuration.

In practice, the WeTab firmware was less cooperative. USB keyboard input worked in firmware and once Linux was running, but not reliably in the first syslinux/extlinux path. That made interactive bootloader selection difficult.

GRUB2 looked promising because it could load USB keyboard support manually. After loading the relevant USB support, keyboard input worked at the GRUB prompt. But that introduced a different problem: the storage view changed, and the expected disk or partition was no longer visible in the way GRUB needed.

That was enough archaeology for one tablet.

The practical solution was simpler: format the internal SSD as VFAT, install syslinux, copy the Cobolito/400 kernel and initramfs there, and boot directly into the appliance with no visible menu. That loses the elegance of multiple selectable appliance entries, but it fits the physical target better. The tablet is not a development workstation. It is supposed to turn into Cobolito/400.

Boot Time

As the development targets table shows, the appliance reached the application quickly on both the laptop sideload path and the WeTab hardware target.

The exact figures should not be treated as a formal benchmark. They were practical proof-of-life measurements: enough to show that the appliance path was narrow, that the initramfs approach worked, and that the physical target did not bury the operator under a normal general-purpose boot sequence.

I was also reasonably proud of that result, considering that my well-engineered Android phone takes rather longer than five seconds to remember it is a phone.

What the Hardware Proved

The hardware work proved that Cobolito/400 was not only a tidy idea inside a VM.

It could boot on physical x86 hardware. It could accept USB keyboard input. It could read input from a USB barcode scanner. It could see a FAT USB stick for export. It could reach the application quickly enough to feel appliance-like. And it could do all of this without starting a normal installed operating system.

That mattered because Cobolito/400 is about the shape of a machine, not only about the software inside it.

A data appliance needs a physical ritual: power on, enter the task, scan or type data, copy records out, shut down. The WeTab was imperfect, mildly stubborn, and occasionally rude, but it gave Cobolito/400 that ritual.

It became a small machine with a job.

Appendix E: Terminal Handling and cttyhack

One of the small but important problems in Cobolito/400 was terminal handling.

At first, the appliance only needed to boot into a shell. At that stage, I was mostly trying to prove that the kernel and initramfs could boot, that the root filesystem was usable, and that basic commands worked.

They did. I could run *ls*, inspect the environment, and feel briefly triumphant.

Then I installed nano as a test.

That was not because nano was meant to be part of the final operator workflow. It was a useful test case because it depends on terminal behaviour and ncurses. If nano worked, I would have more confidence that COBOL screen programs could work later.

Instead, the shell greeted me with:

```
/bin/sh: can't access tty; job control turned off
```

That was the small crack in the floorboards.

The shell could run commands, but it did not have a proper controlling terminal. Running *tty* confirmed the problem: no usable terminal device had been allocated to the process. For simple command execution, this was survivable. For interactive programs that care about terminal state, screen control, and job handling, it was not enough.

A regular Linux system usually starts getty on a console, getty prepares a terminal session, and then the user receives a shell with the expected controlling terminal. Cobolito/400 does not start that whole layer. It has no normal login path, no service manager, and no desire to become a tiny cosplay of a full distribution.

The solution was cttyhack.

cttyhack is known from BusyBox, where it exists as a small utility for starting a process with a usable controlling terminal. The problem was that the BusyBox build shipped by Alpine did not include that applet. Cobolito/400 therefore could not just call the system-provided BusyBox cttyhack.

To keep the appliance small without rebuilding all of BusyBox, I used the implementation from Dima Krasner’s lazy-utils project and kept a local copy in the Cobolito/400 source tree. That made the build reproducible while adding only the tiny piece of terminal plumbing the appliance needed.

In the init script, both the appliance supervisor and the fallback shell are started through cttyhack:

```
if [ -x /data/cobolito-start ]; then
    exec /bin/cttyhack /data/cobolito-start
fi

exec /bin/cttyhack /bin/sh
```

This keeps the init script simple while still giving the interactive application the terminal behaviour it needs.

What cttyhack Solved

cttyhack solved a narrow problem, but it solved the right narrow problem.

Without it, Cobolito/400 could boot, but the interactive environment was not quite right. With it, the appliance could start an operator-facing terminal program directly from the initramfs without adding getty, login handling, or a larger service startup sequence.

That matters because Cobolito/400 is built around subtraction. Every extra component has to justify itself. Adding a conventional login stack would have worked, but it would also have made the appliance feel more like a stripped-down Linux machine than a small dedicated data system.

cttyhack was the opposite: a tiny piece of plumbing that allowed the rest of the system to stay small.

It is not a glamorous part of the project. It is not the kind of thing that appears in a screenshot. But without it, the path from kernel to operator screen would have been much less clean.

Appendix F: Removable Media and cobolito-copy

Cobolito/400 needed a way to move data out of the appliance without turning the operator workflow into a normal Unix administration session.

In my earlier small-system experiments, including PursePC, copying a file to removable media followed the usual ritual: insert a USB stick, become root if needed, mount the filesystem somewhere, copy the file, unmount the filesystem, then remove the device.

That is not difficult, but it is not the feeling I wanted for Cobolito/400.

The older systems that inspired this project had a more immediate removable-media rhythm. Insert the diskette. Copy the data. Remove the diskette. The operator knew there was a physical medium involved, but the workflow did not require thinking in terms of mount points and filesystem plumbing.

For Cobolito/400, I wanted something with that same spirit.

USB Sticks as Modern Diskettes

I still have a USB 3.5-inch floppy drive in the garage, and it would have been a charmingly literal choice. It would also have made the project depend on a device that is now unusual, slow, and mostly decorative outside retrocomputing contexts.

A small FAT or VFAT USB stick felt like the modern equivalent.

It is cheap, common, physically removable, and readable by almost every operating system. A 2 GB or 4 GB USB stick has far more capacity than Cobolito/400 needs, but it still has the right operational meaning: a small piece of media that carries records from one system to another.

The point was not to emulate a diskette mechanically. The point was to preserve the exchange pattern.

Avoiding the Mount Ritual

The problem was how to avoid the usual Unix mount workflow.

Mounting a USB stick is normal on a general-purpose system, but Cobolito/400 is not meant to expose the operator to that machinery. I did not want the user to choose mount points, remember whether a filesystem was mounted, or worry about unmounting manually before removing the device.

That led me back to *mtools*.

mtools is an old set of utilities for working with FAT filesystems without mounting them. I remembered it mostly from floppy-disk workflows, but the important question was whether it would also work directly against other FAT or VFAT block devices, such as USB sticks.

It did.

That made mtools a good fit for Cobolito/400. It allowed the appliance to treat a FAT USB stick as removable exchange media without exposing a full mount/unmount process.

The Operator Flow

The cobolito-copy utility is intentionally simple.

The operator is asked to insert a FAT USB drive and press Enter:

```
Cobolito/400 Copy/Save
Insert FAT USB drive, then press ENTER.
```

The script then scans candidate block devices and looks for filesystems that mtools can read as FAT.

If it finds suitable targets, it presents a small numbered list using the FAT volume label when available:

```
Choose target:

  1) CRUMBUSB (/dev/sdb1)
  2) NO_LABEL (/dev/sdc1)

Target number:
```

The operator chooses the target number.

The script then copies the application data files from the data area to the root of the selected USB device. In the current implementation, the copied files are:

```
/data/*.DAT
```

For the first proof-of-life application, this means files such as CRUMBS.DAT.

The copy is performed with mcopy:

```
mcopy -i "$TARGET" "$SRC_DIR"/*.DAT ::
```

If the copy succeeds, the script reports completion and tells the operator that the USB drive may be removed.

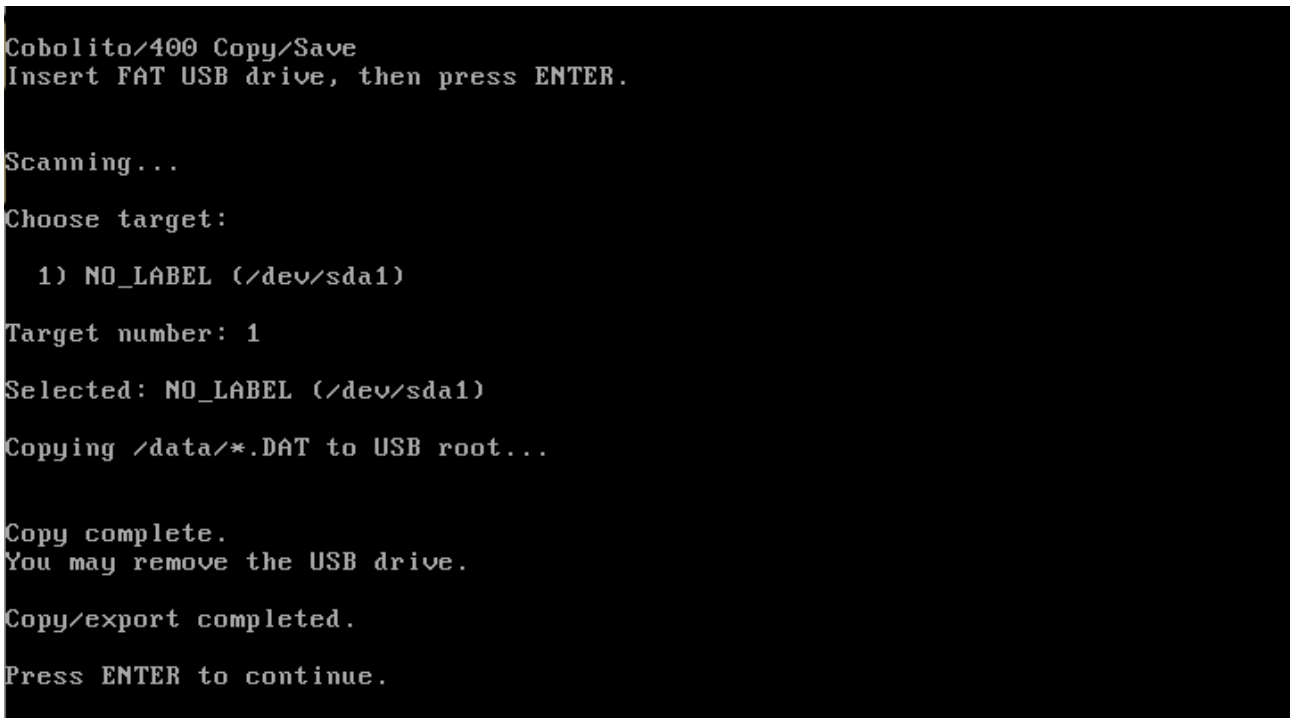

Figure F.1: The cobolito-copy removable-media export workflow.
The Copy/Save utility detects a FAT USB target, copies the appliance data files to the root of the selected device, and tells the operator when the media may be removed. The point of the screen is not sophistication. It is the opposite: a small appliance-level export ritual instead of the usual mount, copy, unmount dance.

Detecting FAT Targets

The script builds a temporary list of candidate FAT devices.

It checks each candidate block device by trying to read it with mdir:

```
mdir -i "$DEV" ::
```

If mdir succeeds, the device is treated as a possible FAT target.

The script scans several common device patterns:

```
/dev/fd0
/dev/sd[a-z]
/dev/sd[a-z][0-9]*
/dev/vd[a-z]
/dev/vd[a-z][0-9]*
/dev/mmcblk[0-9]
/dev/mmcblk[0-9]p[0-9]*
```

Including /dev/fd0 keeps the old floppy-shaped idea alive, even though the practical target is usually a USB stick.

The /dev/sd patterns cover typical USB storage devices. The /dev/vd patterns are useful in virtual-machine environments. The /dev/mmcblk patterns allow for systems where removable or embedded flash storage appears through the MMC block-device naming scheme.

For each readable FAT target, the script extracts the volume label where possible. If no label is found, it uses:

```
NO_LABEL
```

This makes the operator prompt friendlier without requiring a more complex device-discovery system. It is not a full hardware inventory. It is just enough to ask, “which removable thing do you want to copy to?”

No General Filesystem Manager

cobolito-copy is deliberately not a general file manager, backup system, or removable-media framework.

It does not mount filesystems. It does not browse directories. It does not try to support every filesystem. It does not decide where files should go based on rules, metadata, or user profiles. There are no profiles. There is no little policy empire hiding inside the copy script.

It does one thing: copy the appliance data files to a FAT target.

That narrowness is part of the design. The script belongs to the appliance layer, not to a full operating environment. Its job is to provide a visible export action that an operator can understand.

Overwrite Behaviour

The current implementation does not provide a dedicated Cobolito/400 overwrite screen.

That is an important limitation. If a file with the same name already exists on the target media, overwrite handling is left to the behaviour of the underlying mtools command. For a future version, I would prefer the appliance itself to check for existing files and present a clearer operator prompt before copying.

For the first proof of life, I accepted the simpler behaviour. The goal was to prove the removable-media exchange path, not to build a complete media-management interface.

Still, this is one of the places where the current implementation shows its prototype nature.

A real appliance should be boring about overwrites. Boring is good. Boring means fewer surprises at the counter.

Returning to the Supervisor

cobolito-copy is not normally launched by the operator from a shell.

In the appliance workflow, CrumbStation exits with a return code requesting Copy/Save. The supervisor receives that return code, runs cobolito-copy, and then returns the operator to the appliance flow.

That separation is deliberate. CrumbStation records the business event. Cobolito/400 handles the appliance-level export action.

The operator sees one system. Internally, the boundary remains clean.

Limits

The current export mechanism has clear limits.

It expects FAT or VFAT media. It copies data files matching the current application convention. It is designed for simple removable-media exchange, not for general backup, synchronisation, encryption, compression, versioning, or remote transfer.

Those limits are not accidental. Cobolito/400 is an experimental data appliance, and cobolito-copy is the smallest export mechanism that proved the intended workflow.

A later version could add more careful overwrite handling, per-application export manifests, checksums, timestamps, or a clearer confirmation screen. Those would be useful improvements. But they would still need to respect the same basic contract: data should leave the appliance through a visible, understandable operator action.

What the Export Path Proved

The export path proved that Cobolito/400 could move records out without becoming a normal Unix workstation.

That matters because the data appliance idea is incomplete without a way for data to leave. A small local machine that captures records but traps them is not calm. It is just a tiny island with a keyboard.

The Copy/Save workflow gives Cobolito/400 its removable-media contract. Insert the USB stick, choose the target, copy the records, remove the media. Not a literal diskette workflow, but a modern echo of the same operational shape.

The media changed.

The ritual survived.

Appendix G: Appliance/Application Contract

Cobolito/400 is not only CrumbStation.

CrumbStation remains a COBOL application in its own right. Inside Cobolito/400, however, it gains an appliance context. This appendix records the boundary that makes that possible.

The appliance boots, prepares the environment, starts the supervisor, and provides appliance-level services such as Copy/Save, power handling, and the hidden developer shell. The application performs the actual workload: screens, input, records, browsing, and business logic.

The operator sees one small machine.

Internally, there is a contract.

Boundary Between Layers

The boundary is simple:

Layer	Responsibility
Appliance layer	Boot the kernel and initramfs, prepare the runtime environment, load required modules, set terminal handling, start the supervisor, provide Copy/Save and power actions
Supervisor	Start the application, interpret return codes, run appliance utilities, return to the application loop when appropriate
Application layer	Present screens, collect input, write records, browse records, request appliance-level actions through return codes

This keeps the application from becoming responsible for the machine around it.

CrumbStation should not need to know how to copy files to a FAT USB stick, how to power off the system, or how to open a developer shell. Those are appliance behaviours. The application only asks for them.

Environment Variables

The supervisor prepares a small environment before starting CrumbStation.

The most important variables are:

Variable	Purpose
COBOLITO	Tells the application it is running inside the Cobolito/400 appliance environment
STATION	Identifies the station where records are being produced
DEVICE	Identifies the specific appliance or terminal
LD_LIBRARY_PATH	Points the runtime linker at the application-local library area

The COBOLITO variable allows the same application to behave differently inside and outside the appliance. When CrumbStation is running as a normal program on a development system, appliance-only functions can be hidden or disabled. When it is running inside Cobolito/400, those functions become part of the operator screen.

LD_LIBRARY_PATH looks like ordinary Unix plumbing, and technically it is. But in Cobolito/400 it also fits the smaller version of the TULIP/400 library idea.

In TULIP/400, a library is a place where related objects belong together. Cobolito/400 reduces that idea almost to its minimum. The data area is effectively the appliance's only application library. It contains the application, the runtime libraries needed by that application, the supervisor script, configuration material, and the data files produced by the workload.

That is why the runtime linker is pointed there. The appliance image provides the small base system; the data area carries the application world.

This was important because I did not want CrumbStation to become trapped inside one appliance image. It should remain a COBOL application first, and an appliance application when the appliance context is present.

Boot-Time Station and Device Identity

The station and device values are provided through the bootloader command line.

A syslinux or extlinux-style entry can include appliance-specific parameters like this:

```
LABEL cobolito
  KERNEL vmlinuz-lts
  INITRD cobolito
  APPEND quiet loglevel=0 rootdelay=0 nomodeset autosuspend=0 init=/init
cobolito.station=PARIS cobolito.device=HPWRK01
```

The important part is at the end of the APPEND line:

```
cobolito.station=PARIS cobolito.device=HPWRK01
```

The supervisor reads those values from the kernel command line, exports them as environment variables, and starts the application.

CrumbStation then writes them into each record.

That gives exported data a useful origin. A record is not only something that happened at a time. It happened at a station, on a device. If several appliances later feed the same back-office process, those fields become part of the data contract.

This mechanism is intentionally simple. There is no configuration database, no network registration service, and no grand ceremony involving certificates, agents, or a dashboard wearing too much cologne. For the first appliance, the bootloader line was enough.

Return Code Contract

The application requests appliance-level actions by exiting with specific return codes.

Key	Return code	Meaning	Supervisor action
F3	0	Exit application normally	Return to the supervisor/service flow
F5	10	Copy/Save requested	Run the removable-media export utility
F12	20	Power action requested	Show the power menu
F9	90	Developer shell requested	Open the hidden developer shell

The operator does not see return codes. The operator sees function keys.

The return codes are the internal agreement between the COBOL application and the appliance supervisor. CrumbStation does not copy to USB directly. It exits with the Copy/Save return code. The supervisor receives that code and runs the appliance-level export utility.

The same applies to power handling. CrumbStation does not directly shut down the machine. It requests the power menu. The supervisor owns the actual appliance action.

Why Return Codes?

COBOL could have called system commands directly.

For example, CrumbStation could have used a system call to run a copy script or invoke a shutdown command. That would have worked, at least on one system.

But it would also have made the application more system-dependent. The COBOL program would start to know too much about the operating environment around it. That is exactly what I wanted to avoid.

Return codes are blunt, old-fashioned, and very useful here.

They let the application say: *"I am done for now, and I am requesting action 10"*.

The supervisor decides what action 10 means on this appliance.

That keeps CrumbStation portable. On a normal development system, return code 10 can simply be observed or ignored. Inside Cobolito/400, it becomes Copy/Save. In another appliance, it could mean something else, or it could be mapped differently by a different supervisor.

The COBOL application remains small. The appliance remains responsible for appliance things.

Supervisor Loop

The supervisor runs CrumbStation in a loop.

At a high level, the flow is:

1. prepare environment
2. start CrumbStation
3. read return code
4. perform requested appliance action
5. return to CrumbStation or exit

This is not complicated, and that is a strength.

The supervisor is the hinge between the application and the appliance. It gives the application a way to request actions without embedding operating-system details inside the COBOL code.

The first version handles the known return codes and treats unexpected return codes as errors or service conditions. That behaviour is useful during development because it makes surprises visible instead of quietly pretending that everything is fine. Small systems should not be mysterious when they are confused.

Power Menu

The power menu is deliberately outside the COBOL application.

When CrumbStation requests a power action, the supervisor presents the appliance-level choices: shutdown, reboot, or cancel. Before shutdown or reboot, the workflow can ask whether data should be copied to removable media first.

That reminder matters because the appliance stores local records. Powering off should not quietly strand useful data inside the machine.

The power menu therefore belongs to the appliance contract, not to the application logic. CrumbStation can ask for power handling, but Cobolito/400 owns the ritual around it.

Developer Shell

The hidden developer shell exists for one reason: early appliances break.

During development, a small escape hatch is invaluable. If the application fails, if the data area needs inspection, or if a device behaves strangely, the developer shell provides a way to look inside the running appliance without rebuilding everything blindly.

This is intentionally not an operator feature. It is not advertised on the normal screen, and it should not become part of the ordinary workflow. It is a maintenance hatch, not a user interface.

Tiny appliances still need a way for the mechanic to lift the lid.

What the Contract Keeps Separate

The appliance/application contract keeps Cobolito/400 from becoming a tangle.

That is the point of the contract: the application owns the records, and the appliance owns the machine. CrumbStation does not need to know how Cobolito/400 boots, copies data, opens a developer shell, or shuts down. It only needs to know that, when it is running inside the appliance, certain function keys can request appliance-level actions.

The supervisor sits between the two layers, passing small messages in a deliberately old-fashioned way: environment variables going in, return codes coming out.

Appendix H: Data Format Contract

This appendix describes the first Cobolito/400 data contract.

For the proof-of-life application, CrumbStation writes its records to a COBOL line sequential file called CRUMBS.DAT. That choice was deliberate.

I did not choose a generic sequential file. I chose *line sequential* because, with GnuCOBOL on Unix-like systems, each record is written as a line with a line ending. That made the file immediately inspectable with ordinary tools such as nano, less, or cat. Without those line endings, the data could still be sequential, but it would be much less pleasant to inspect by eye: one long record stream, technically valid perhaps, but not exactly friendly.

The file is not clever. That is a compliment.

It is one fixed-layout COBOL record per line. Each line represents one event captured by the appliance: a loyalty action, a parcel scan, or another small local transaction. The structure is defined by the COBOL record layout, and any receiving system that understands that layout can parse the file.

The application writes the record. The contract is what lets it travel.

COBOL Record Layout

The CrumbStation record is defined as:

```
01 CRUMB-RECORD.
   05 CRB-DATE             PIC 9(8).
   05 CRB-TIME             PIC 9(8).
   05 CRB-STATION          PIC X(20).
   05 CRB-DEVICE           PIC X(20).
   05 CRB-TYPE             PIC X.
   05 CRB-CODE             PIC X(50).
   05 CRB-QTY              PIC 9(3).
```

The total logical record length is 110 characters, plus the line ending used by the host environment.

Field	Picture	Length	Meaning
CRB-DATE	9(8)	8	Date in YYYYMMDD form
CRB-TIME	9(8)	8	Time in HHMMSScc form
CRB-STATION	X(20)	20	Station or site identifier
CRB-DEVICE	X(20)	20	Device or appliance identifier
CRB-TYPE	X	1	Event type
CRB-CODE	X(50)	50	Scanned or entered code
CRB-QTY	9(3)	3	Quantity or points value

The time field uses HHMMSScc, where cc represents hundredths of a second. It is not milliseconds. That small distinction matters because a field that looks precise can easily be over-interpreted by a future reader, including a future version of me with less coffee.

Example Records

A few example records look like this:

```
20260625082033 55PARIS             HPWRK01             S937473278                                017
20260625082043 90PARIS             HPWRK01             E7373737                                  001
20260625082048 95PARIS             HPWRK01             E39399337671                              004
20260625082052 30PARIS             HPWRK01             S234986                                   001
```

The records are intentionally plain.

The date and time identify when the event was captured. The station and device fields identify where it was captured and by which appliance. The type field identifies the kind of event. The code field carries the scanned or entered value. The quantity field gives a small numeric value for events that need one.

This is not a database schema trying to win a beauty contest. It is a handoff contract.

Character and Layout Assumptions

The first CrumbStation files are written as plain text line sequential records.

That means the receiving system must pay attention to three things:

Concern	Practical meaning
Record layout	Fields are positional, not delimited
Character encoding	The first examples use simple ASCII-compatible content
Line endings	The file may need conversion depending on the receiving system

Because the fields are positional, spaces are meaningful. CRB-STATION, CRB-DEVICE, and CRB-CODE are padded to their defined lengths. A receiver should not treat the file as comma-separated, tab-separated, or free-form text.

This is one reason I like the format. It looks simple, but it still has a contract. The line is plain text, but it is not vague.

Line Endings

On Unix-like systems, GnuCOBOL line sequential files normally use LF line endings.

That matters when moving the file to non-Unix systems. The record layout is simple, but the receiving system still has to handle line endings correctly. In practice, an import step may need to convert LF to the target system's expected record representation, or use a transfer method that performs the right conversion.

This became visible during the IBM i import experiment.

The line sequential choice helped because each Cobolito/400 record was already separated as one logical line. The receiving side still needed the correct target file definition and transfer behaviour, but the source file had the right record rhythm: one line, one logical record.

Import into IBM i

The most satisfying experiment was importing Cobolito/400 data into IBM i.

That was important to me because IBM i is a record-oriented environment. If the file contract was real, then the records should be able to leave Cobolito/400 and become ordinary structured data somewhere else.

The receiving side was an IBM i Physical File created through DDS from the COBOL copybook structure. The fields matched the Cobolito/400 record layout: date, time, station, device, type, code, and quantity.

The import path was deliberately simple:

```
Cobolito/400 CRUMBS.DAT
  ↓
transfer as text
  ↓
IBM i Physical File with matching DDS layout
  ↓
record inspection through IBM i tools
```

Once the Physical File existed, the file could be transferred into it and inspected as normal IBM i data. The records were visible with tools such as DSPPFM, RUNQRY, and STRDFU.

That last one made me smile more than it probably should have.

STRDFU could read and manipulate records that had started life inside Cobolito/400, written by a tiny COBOL application in an Alpine initramfs, copied through a removable-media workflow, and then imported into IBM i as ordinary business records.

That is the kind of round trip this project was built to prove.

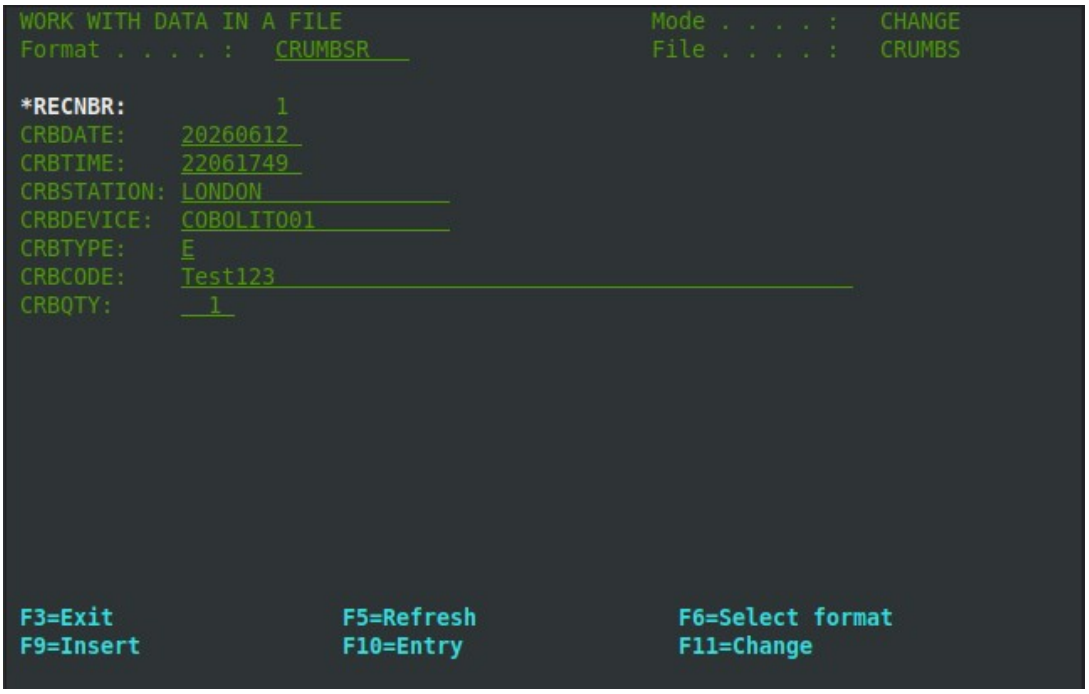

Figure H.1: Cobolito/400 records imported into an IBM i Physical File and viewed through STRDFU.
The screenshot shows CRUMBS.DAT after import into a matching record-oriented IBM i structure. The important part is not the screen itself, but the fact that the records are now ordinary fielded data on another system.

The import still required a correctly defined target Physical File and a suitable transfer step. There was no magic involved, which is healthy. The point is that no Cobolito/400 back end was needed on IBM i. The file contract was enough.

Import into TULIP/400

The same idea also fits the TULIP/400 direction.

In the planned TULIP/400 flow, CrumbStation records can be imported into a SQLite-backed environment through a GnuCOBOL and GixSQL path. The importer can dispatch records according to the event type and place them into a more queryable structure.

This is not only a theoretical path. I experimented separately with GnuCOBOL, GixSQL, and SQLite, and kept small implementation examples for that combination. Those notes and examples are listed in the Sources and References rather than repeated here.

That receiving system is different from IBM i, but the principle is the same. Cobolito/400 captures the records locally. The receiving system knows the contract and decides what to do with them.

The appliance does not need to carry the future back office inside itself.

What the File Contract Proved

The file contract proved one of the central ideas of Cobolito/400: a tiny appliance can write records that remain useful after they leave the machine.

That sounds modest, but it matters. Many systems make data portable only through the original application, a service API, a database engine, or a stack of assumptions hiding under a pleasant export button. Cobolito/400 tries to do something smaller and more explicit.

The record layout is known, the file can be inspected, and another system can import it once it understands the contract. The receiving system may still need conversion, transfer handling, or a matching target structure. That is normal. The important part is that the data is not trapped inside the appliance.

For me, this was the emotional centre of the prototype. Not the fastest boot, not the smallest initramfs, not even the COBOL screen. The important moment was seeing the records leave the tiny appliance world and reappear as structured data somewhere else.

In a project inspired by data-entry systems, IBM i, DDM, and TULIP/400, that mattered more than any boot-time figure.

Appendix I: Custom GnuCOBOL Build

CrumbStation is a GnuCOBOL application, but Cobolito/400 does not use the full distribution-packaged GnuCOBOL runtime in its first appliance image.

That choice was practical rather than ideological.

On normal systems, such as Linux Mint or FreeBSD, CrumbStation can be built and run with the GnuCOBOL packages provided by the distribution or ports collection. Those environments are general-purpose operating systems, and carrying the normal runtime dependencies is not a problem.

Cobolito/400 is different. It is a small initramfs-based appliance, so every library and runtime dependency included in the image becomes part of the appliance contract.

Why Not the Alpine Edge gnucobol3 Package

GnuCOBOL 3 was not available in the Alpine LTS main or community repositories I was using for Cobolito/400. There was, however, a gnucobol3 package available from the Alpine Edge repository.

That package was attractive at first because it provided a ready-made GnuCOBOL environment. I did not switch the appliance base to Alpine Edge, but I did consider using the Edge gnucobol3 package as a convenient way to obtain the compiler and runtime.

However, the packaged runtime included support for features that Cobolito/400 did not need in its first version, including DBM-backed indexed files, XML support, and JSON support. Those features are useful in a general-purpose GnuCOBOL installation, but they also pull in additional runtime dependencies.

Including those dependencies would have made the image larger and less focused without helping the workload.

Building Only What the Appliance Needed

For the Cobolito/400 appliance version, I built GnuCOBOL with unnecessary features disabled.

The goal was not to create a special or incompatible COBOL dialect. The goal was to keep the runtime close to what the appliance actually needed.

CrumbStation’s file contract was deliberately simple: COBOL line sequential files, with a fixed-layout record inside each text line. The application needed COBOL program execution, line sequential file handling, screen support through the terminal and ncurses

path, and a small libcob runtime inside the appliance. It did not need DBM-backed indexed files, XML parsing, JSON handling, or every optional GnuCOBOL feature.

This fits the appliance principle used elsewhere in Cobolito/400: include what the workload needs, not every possible feature the ecosystem can provide. A small appliance should not be an argument between all available options. It should be a decision.

Static and Dynamic Linking

I also experimented with static linking.

In theory, a fully static CrumbStation binary would be attractive for an appliance. The application could carry more of its runtime with it, and the surrounding filesystem could be smaller or simpler.

In practice, that path became more complicated than it was worth for the first version. Making GnuCOBOL applications fully static can involve more than passing a different linker option. The compiler, libcob, terminal libraries, and other runtime pieces all have to line up correctly.

Rather than spending the project budget on static-linking archaeology, I kept the application dynamically linked and reduced the runtime dependencies instead.

That means the CrumbStation binary is not the whole application environment by itself. The required libcob runtime still has to be present in the appliance image.

For Cobolito/400, that was acceptable.

Runtime as Part of the Library Contract

Keeping libcob in the appliance also fits the broader TULIP/400 idea of a small application library.

The COBOL program is not treated as a magical self-contained object. It has a known runtime environment, and that environment is deliberately included in the appliance. The important part is that the environment remains small, understandable, and aligned with the application's real needs.

In other words, CrumbStation on Linux Mint or FreeBSD can use the normal system GnuCOBOL runtime. CrumbStation on Cobolito/400 uses a trimmed runtime included in the appliance image. This keeps development convenient on ordinary systems while keeping the appliance image lean.

What the Runtime Choice Kept Small

The runtime choice is another example of the same design rule used throughout Cobolito/400.

A general-purpose system should provide general-purpose facilities. A small appliance should carry only the facilities needed by its task.

For CrumbStation, that meant keeping COBOL, line sequential files, terminal screens, and the minimal runtime support required to make them work. Everything else could stay outside the appliance, waiting for a future workload that actually needs it.

Not every dependency needs to be invited to tea.

Appendix J: COBOL Notes Learned While Building CrumbStation

I learned a great deal while building Cobolito/400.

Part of that was technical in the appliance sense: initramfs, bootloaders, terminal handling, USB modules, removable media, and the small practical edges of turning a design idea into something that actually boots.

But the part where I probably learned the most, in a direct programming sense, was COBOL.

Before CrumbStation, I had experimented with COBOL. I had written basic command-line programs and simple screen-input examples. CrumbStation took that further. It forced me to deal with files, screens, function keys, subprograms, and implementation-specific runtime behaviour in the context of a complete proof-of-life application.

This appendix is not intended as a COBOL tutorial. It is a set of notes about the things I learned while building CrumbStation, especially the parts that were not obvious at the beginning.

It was also a small confidence boost. Cobolito/400 is not rocket science, but it reminded me that I can still take a trail from research, through design, through hardware experiments, through code, and arrive at a working system.

That mattered.

Screen Handling

COBOL screen handling was the next large lesson.

Creating a basic input mask was one thing. Creating a useful operator interface was another.

The main CrumbStation screen is fairly simple: enter or scan a code, choose an action, enter a quantity when relevant, and write the record. That screen is close to a form. It has fields, labels, prompts, and function keys. It is important, but its behaviour is still narrow.

The browse screen was a different problem.

That became the CRUMBLIST module, and it taught me that a text user interface is an architecture of its own. It is not only a matter of placing fields on a screen. The program has to decide what the operator can see, which actions are available, where those actions apply, what feedback is shown, and how the screen state changes after each operation.

For the browse screen, I had to think about how many records fit on one screen, how to display rows clearly, how to associate an operator action with a specific row, how to mark one or more records for deletion, how to paginate without confusing the operator, how to show useful messages, how to return cleanly to the main screen, and how to avoid exposing appliance functions in the wrong place.

That was much more than drawing a list.

A browse screen has to behave like a small controlled workspace. The operator needs to understand where they are, what they can do, and what has happened after they do it. If a record is marked for deletion, the screen has to make that visible. If the end of the file has been reached, the available function keys should reflect that. If an action is not meaningful in that context, it should not be shown.

This is why I intentionally kept the browse screen as a service page rather than as another full appliance command centre. It is there to inspect records and correct mistakes, not to become the main operating mode of Cobolito/400. The appliance-level actions belong on the main screen and in the supervisor flow.

The sequential file underneath made the browse screen harder, but the larger lesson was about interaction design. Even in a small COBOL program, the TUI becomes its own design problem: fields, rows, messages, function keys, pagination, and operator expectations all have to fit together.

It is easy to underestimate that.

I did.

Function Keys and CRT STATUS

Once the screens existed, the next practical question was how the program should recognise operator actions.

In CrumbStation, function keys are not decorative labels at the bottom of the screen. They are part of the control flow. F3 exits, F4 opens the browse screen, F5 requests Copy/Save, F12 requests the power menu, and the hidden F9 maintenance path opens the developer shell. For those keys to matter, the COBOL program has to receive them as input events and turn them into decisions.

In GnuCOBOL, that meant using CRT STATUS.

In the configuration section, the console can be named and a CRT status field can be defined:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
   CONSOLE IS CRT.
   CRT STATUS IS COB-CRT-STATUS.
```

With that in place, the program can inspect COB-CRT-STATUS after an ACCEPT and react to function keys. For example:

```
IF COB-CRT-STATUS = COB-SCR-F3
   MOVE 'N' TO WS-RUN-FLAG
   EXIT PARAGRAPH
END-IF.
```

That was the useful COBOL lesson. A function key is not just a visual convention. It has to be wired into the screen accept logic, reported through a status field, and handled explicitly by the program.

Once that worked, the screen stopped being only a form. It became a small front panel.

That was the moment CrumbStation started to feel like the kind of application I wanted.

The Input Field Detail

CRT STATUS alone was not enough.

A screen cannot live on function keys alone. To make function-key handling work reliably, the screen needed at least one input field. In practice, an ACCEPT on a screen with a USING field gave the program the interaction point it needed.

That input field does not always have to be meaningful to the operator. It can be a real field, such as the code or quantity field, or in some designs it could be a small dummy field placed somewhere unobtrusive.

The important lesson was that screen structure matters. The function key is not floating in space. It is part of the screen input operation.

That was another small but important practical discovery.

Subprogram State and PROGRAM-ID IS INITIAL

CrumbStation was built from two COBOL programs. The main screen lived in crumbstation.cob, while the browse/delete functionality lived in crumblist.cob.

This separation was useful during development. The browse module took a lot of work, especially around the user interface, pagination, sequential-file handling, rewinding, and deletion logic.

At first, I could have connected the modules using a process-level contract similar to the supervisor return codes. But I wanted a single binary, so I built the application with CRUMBLIST as a called COBOL program.

In theory, this was straightforward.

In practice, it produced one of the more confusing bugs of the project.

The first time I pressed F4, the browse functionality opened correctly. I could inspect the records and return to the main screen. But when I pressed F4 again, nothing happened.

I spent days trying to understand why.

Eventually, with debugging hooks in the code, I found the cause. When the called program returned with GOBACK, its working-storage values were still in the state they had reached during the previous call. In my case, the flag used to exit the browse loop, WS-RUN-FLAG, had been set to N. When the program was called again, that value was still there, so the loop did not behave as I expected.

The solution was to mark the called program as initial:

```
PROGRAM-ID. CRUMBLIST IS INITIAL.
```

With IS INITIAL, the program's storage is reset to its initial state when the program is entered again. That fixed the problem.

It was a small line, but it carried a large lesson: COBOL subprograms have state, and that state matters.

Line Sequential in Practice: Browsing and Deleting

Record deletion was another place where the file contract became visible.

By this point in the essay, the reason for choosing a COBOL line sequential file should already be clear. I will not repeat the whole argument here. The practical COBOL lesson was narrower: once CRUMBS.DAT was simple, portable, and easy to inspect, the

application had to do the work that a database table or indexed file might otherwise have hidden.

In CrumbStation, records can be marked for deletion from the browse screen. But CRUMBS.DAT is a line sequential file. It is not a database table, and it is not an indexed file with a deletion mechanism. There is no tiny trapdoor under a record where the program can quietly make it disappear.

So deletion is implemented as a rebuild.

The program creates a temporary file, CRUMBS.TMP. It writes all records except the ones marked for deletion. Once the temporary file has been created successfully, the application replaces the original data file with the rebuilt version.

In the GnuCOBOL implementation, the rename operation uses a built-in subroutine:

```
CALL "CBL_RENAME_FILE" USING "CRUMBS.TMP" "CRUMBS.DAT".
```

This is implementation-specific. It works for this GnuCOBOL proof of concept, but it should not be mistaken for a universal COBOL portability guarantee.

The lesson was not that deletion is difficult in the abstract. The lesson was that simple file contracts often move complexity into visible application logic. The data file becomes easier to copy, inspect, archive, and import, but the program must explicitly handle behaviours that a larger database or indexed runtime might provide more directly.

The file stayed simple.

The program paid the bill.

Documentation Friction

Many of these discoveries were possible because information exists. But the information was not always easy to find.

Practical GnuCOBOL knowledge is often scattered: manuals, examples, mailing-list posts, forum answers, old snippets, source trees, and trial-and-error notes. The same was true for function keys, screen handling details, subprogram behaviour, and implementation-specific runtime calls.

That does not make GnuCOBOL bad. It simply means that building a real small application required a bit of archaeology.

Perhaps that is fitting for this project.

What I Took From It

I am not pretending to be an experienced COBOL developer now. I may never be one in the traditional sense. But building CrumbStation taught me much more than writing isolated examples would have done.

I learned how file organisation choices affect the whole application. I learned that line sequential files are wonderfully portable, but not magically convenient. I learned that screen programs require careful thinking about interaction, not only fields. I learned that function keys can make a COBOL screen feel like a real operator interface. I learned that called programs can preserve state in ways that matter. I learned that simple deletion from a sequential file is not really deletion, but reconstruction.

Most importantly, I learned that COBOL still makes sense when the problem is shaped around records, files, screens, and business events.

CrumbStation is tiny. It is only a demonstration application. But it was enough to teach me the practical texture of COBOL: not as an abstract old language, but as a way to build a small record-oriented application that can sit inside an appliance and do one clear job.

Appendix K: CrumbStation Operator Walkthrough

This appendix shows CrumbStation from the operator's point of view.

The main essay describes why the application exists, and the earlier appendices describe the appliance boundary, the return-code contract, the data file, and the export path. This section is simpler: it shows what the small machine looks like when it is being used.

CrumbStation is intentionally modest. It is not a complete shop system, a polished product, or a back-office application. It is a proof-of-life workload for Cobolito/400: a small COBOL screen application that can collect local events, write fixed-layout records, browse what has been entered, and request appliance-level actions such as USB export or power control.

The screenshots are included less as decoration than as evidence. They show the operator contract in its visible form: fields, function keys, prompts, lists, confirmations, and the quiet little ritual of entering data into a local machine.

Note on Screenshots

The screenshots in this appendix were taken from Cobolito/400 running under VirtualBox.

The appliance itself also runs on physical hardware, including the WeTab target described earlier, but the physical machines are intentionally minimal and do not include screenshot tooling. I tried taking photographs of the laptop and WeTab screens, but reflections, lighting, and display quality made the details harder to read.

The VirtualBox screenshots are therefore used for clarity. They show the same CrumbStation application and appliance workflow, but in a form that is easier to reproduce and understand on the page.

Main Entry Screen

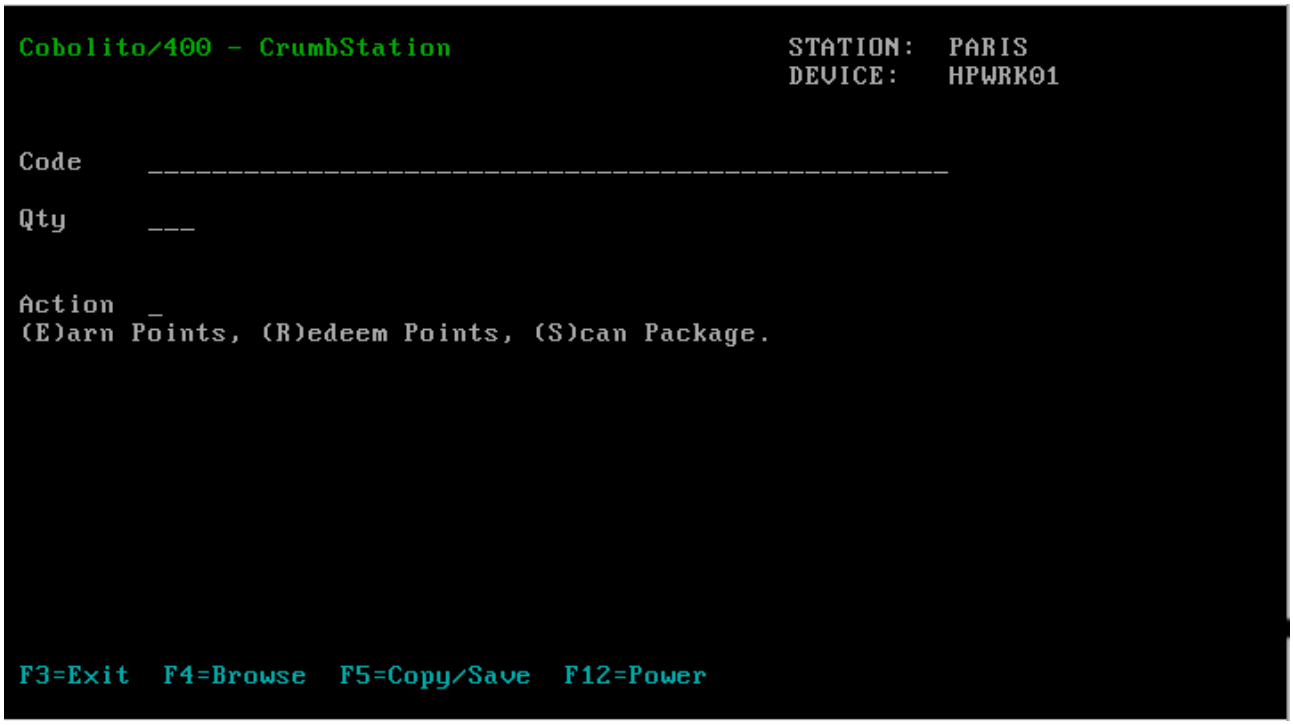

Figure K.1: CrumbStation main entry screen at appliance startup.
This is the primary operator screen shown when Cobolito/400 boots into CrumbStation. The operator enters or scans a code, selects the event type, and, when relevant, enters a quantity. The station and device values in the upper-right corner, PARIS and HPWRK01 in this example, are passed through the bootloader and exported into the application environment.

The visible function keys expose the main appliance workflow: F3 Exit, F4 Browse, F5 Copy/Save, and F12 Power. The hidden F9 developer shell is intentionally not shown. It is a maintenance hatch, not part of the operator contract.

In normal use, the operator scans or enters a loyalty-card code or parcel identifier, chooses the action, and records the event. The quantity field is mainly meaningful for loyalty-point operations such as earn or redeem; for parcel scans, the most relevant information is the code itself, together with the automatically recorded date, time, station, and device identity.

The data is saved when the correct action is specified and Enter is pressed. No database screen appears. No desktop opens. No small cloud-shaped goblin asks for a login. The machine records the event and returns to its task.

Browse Screen

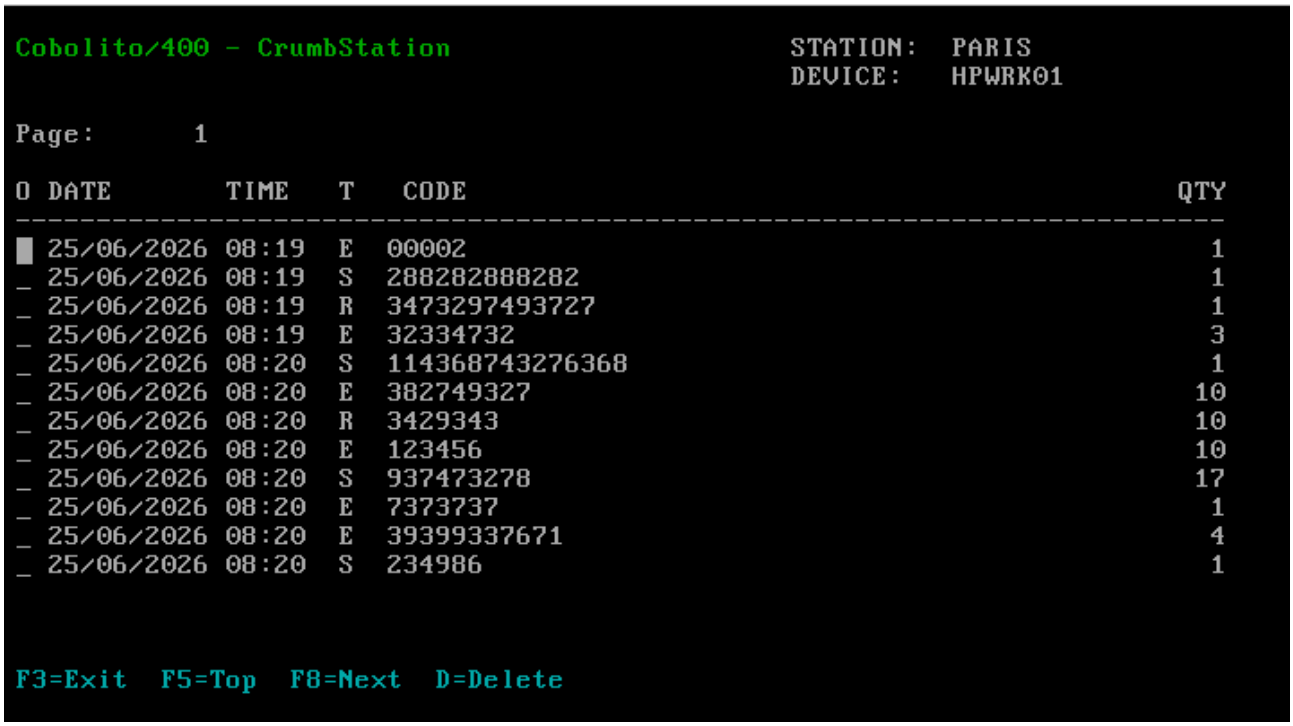

Figure K.2: CrumbStation browse screen.
This screen allows the operator to inspect the contents of CRUMBS.DAT and, if necessary, mark one or more records for deletion. The same station and device identity is shown in the upper-right corner.

F8 moves to the next page, while F5 returns to the top of the file. The current page number is shown, but there is no total page count. This is one of the places where the trade-offs of the data contract become visible: as discussed earlier, the line sequential file is simple and portable, but it is still read as a sequential stream rather than through an indexed structure.

Records can be marked with D for deletion. If deletion is confirmed, CrumbStation rebuilds the file without the marked records. F3 returns to the main screen.

That limitation was accepted deliberately: browse/delete was meant as a small service function for inspection and correction, not as the main operating mode of the appliance.

Power Menu Screen

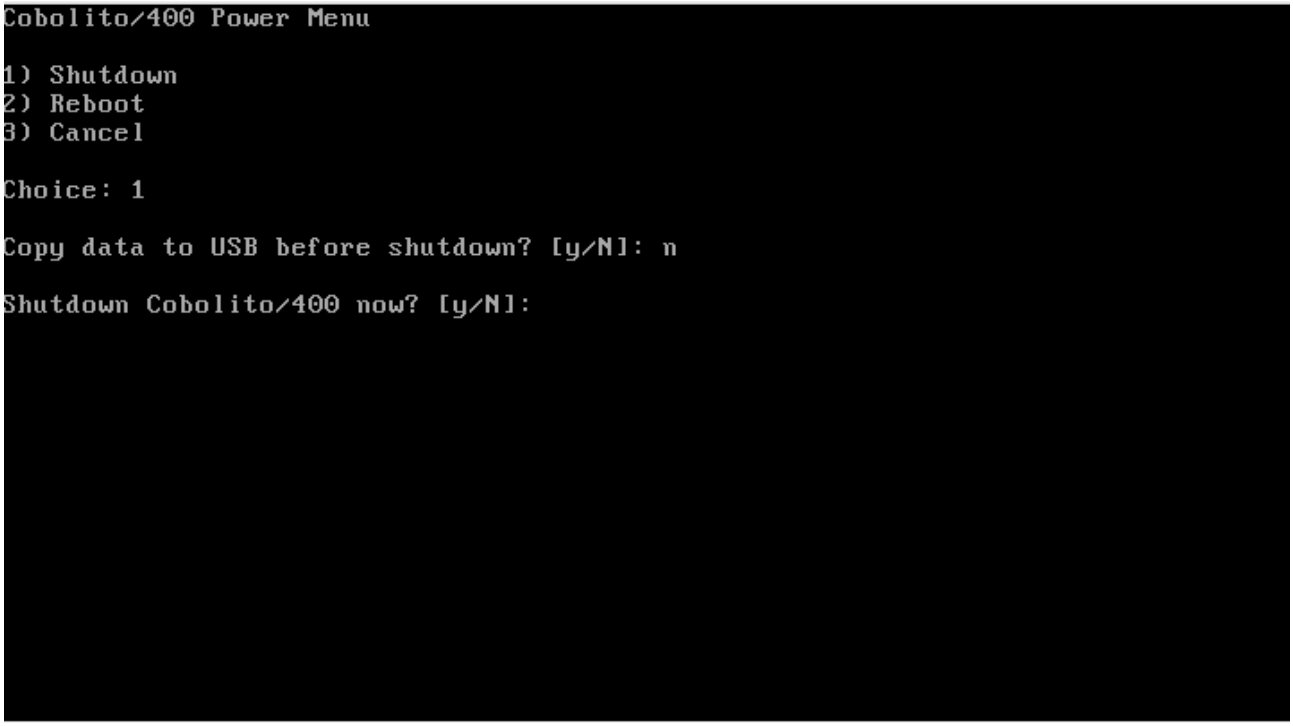

Figure K.3: Power menu with copy reminder.
When the operator requests a power action, Cobolito/400 presents a small power menu with shutdown, reboot, and cancel options. If shutdown or reboot is selected, the appliance asks whether data should be copied to USB first.

This keeps the export ritual close to the power-off workflow, reducing the risk of walking away from local records that have not yet left the machine.

The power menu is deliberately small. It is not a system administration console. It is the final operator decision point: save if needed, shut down if done, cancel if the machine should return to work.

That is the whole operator loop in miniature: enter records, inspect them when necessary, copy them out, and power the appliance down cleanly.

Sources and References

This essay is based on a mixture of IBM manuals, IBM Systems Journal material, archive pages, preserved product literature, secondary reference material, personal implementation notes, and private recollections. Where possible, I have preferred primary or near-primary sources.

IBM 3740 Data Entry System

IBM. *IBM 3740 Data Entry System: 3741 Data Station, 3742 Dual Data Station, Reference Manual*. GA21-9151-0. January 1973.

IBM. *IBM 3740 Data Entry System: System Summary and Installation Manual / Physical Planning*. GA21-9152-3. May 1979.

Wikipedia contributors. “IBM 3740.” *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/IBM_3740. Accessed June 2026.

Wikipedia contributors. “Unit record equipment.” *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Unit_record_equipment. Accessed June 2026.

IBM 5280 Distributed Data System

IBM. *IBM 5280 Distributed Data System: General Information Manual*. GA21-9350-3. 1981.

IBM. *IBM 5280 Distributed Data System: System Concepts*. GA21-9352-1. June 1980.

IBM. *IBM 5280 Distributed Data System: Functions Reference Manual*. GA21-9353-0. February 1980.

IBM. *IBM 5280 Distributed Data System: Operator’s Guide*. GA21-9364-1. February 1981.

IBM. *IBM 5280 Distributed Data System: User’s Setup Procedures*. GA21-9365-2. July 1982.

IBM. *IBM 5280 Distributed Data System: System Control Programming Reference*. GC21-7824-1. May 1981.

IBM. *IBM 5280 Distributed Data System: DE/RPG Reference*. SC21-7787-2. June 1981.

IBM. *IBM 5280 Distributed Data System: Utilities Reference*. SC21-7788-2. April 1981.

IBM. *IBM 5280 Distributed Data System: Sort/Merge Reference*. SC21-7789-0. January 1980.

IBM. *IBM 5280 Distributed Data System: Assembler Manual*. SC21-7790-0. January 1980.

IBM. *IBM 5280 Distributed Data System: Communications Reference*. SC34-0247-1. June 1980.

IBM. *IBM 5280 COBOL Programmer's Guide*. SL23-0032-1. February 1981.

IBM. *IBM 5280 Distributed Data System*. Product brochure / announcement material. G580-0274-00.

IBM. *IBM 5280 Distributed Data System*. Additional preserved product literature, foldout material, and brochure material.

Computerworld. "Japan's NTT Negotiating With IBM for 5280s." October 13, 1980, p. 95. Contemporary trade-press report on negotiations involving IBM 5280 equipment.

Computerworld. "Business Applications Geared to IBM 5280." October 13, 1980, p. 96. Contemporary trade-press item noting business application packages for the IBM 5280 Distributed Data System.

Wikipedia contributors. "IBM 5280 Distributed Data System." *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/IBM_5280_Distributed_Data_System. Accessed June 2026.

IBM Rochester, System/3x, System/38, AS/400, and IBM i Context

IBM Archives. *Rochester Chronology*, page 4. Used for historical context around IBM Rochester, General Systems Division, and the IBM 5280 announcement.

IBM. *City of Santa Cruz System/38 Proposal*. System overview and proposal material. Used as background for the System/38 workstation, database, shared data, and business-system vocabulary.

James Cooper, Nancy Stern and Robert A. Stern. *Programming in Cobol/400, 2nd Edition*. John Wiley & Sons, Inc. 2002. ISBN 0471418463.

Wikipedia contributors. "IBM System/3." *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/IBM_System/3. Accessed June 2026.

Additional IBM product and system literature relating to System/3x, System/38, AS/400, and IBM i, used as background for the discussion of records, screens, files, jobs, and business-system vocabulary.

IBM Distributed Data Management

IBM Systems Journal. “Inside IBM’s Distributed Data Management Architecture.” *IBM Systems Journal*, Vol. 31, No. 3, 1992. DOI: 10.1147/sj.313.0459.

IBM. *Distributed Data Management Architecture: General Information.* GC21-9527-02. Research pointer only: IBM lists this as part of the DDM architecture book set, but I was unable to locate and consult a copy while preparing this essay.

Wikipedia contributors. “Distributed Data Management Architecture.” *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Distributed_Data_Management_Architecture. Accessed June 2026.

Record-Oriented and Data-Centred Systems

Digital Equipment Corporation. *RMS-11 User’s Guide*. Order No. AA-D538A-TC. March 1979.

Hewlett-Packard. *HP 3000 Computer Systems: TurboIMAGE Data Base Management System Reference Manual*. Part No. 32215-90050. December 1985.

Trinetta, Vittorio, and Mario Capurso. *Il manuale del dBASE III Plus*. Gruppo Editoriale Jackson. Italian edition. September 1987. ISBN 8879567931.

Documentation and preserved material relating to VMS Record Management Services, HP 3000 TurboIMAGE, PickOS, MAPPER, dBase III+, DBF files, and other record-oriented or data-centred systems. These were used as background context rather than as the primary historical focus.

MAPPER / Business Information Server

Schlueter, Louis. *Evolutionary Computing Application Design By Users: Next Generation Computing Users Design Major Applications Proven By a Worldwide $3 Billion Customer Base*. Independently published, 2023. ISBN 9798371014191.

Business Information Server / MAPPER historical and promotional material. https://sites.google.com/view/businessinformationserver/home. Accessed June 2026.

Consul 2715

Consul 2715. Preserved documentation / product material. Used only as a possible historical footnote around IBM 5280-compatible or 5280-inspired systems.

NoSQL / Carlo Strozzi

Wikipedia contributors. “Strozzi NoSQL.” *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Strozzi_NoSQL. Accessed June 2026.

Strozzi, Carlo. “NoSQL Home Page.” http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page. Accessed June 2026.

Personal recollection and professional context relating to Carlo Strozzi and the original NoSQL project.

Author’s Own Project Notes and Implementation Sources

Stella, Tara. “Data First, Programs as Guests.” 2026. https://www.tara.sh/posts/2026/2026-02-16_data_first/

Stella, Tara. “The Hermit Project: Summary.” 2024. https://www.tara.sh/posts/2024/2024-04-18_hermit_summary/

Stella, Tara. “End of Summer Updates.” 2025. https://www.tara.sh/posts/2025/2025-09-14_end_summer_updates/

Stella, Tara. “GixSQL, COBOL and FreeBSD”. 2026. https://www.tara.sh/posts/2026/2026-06-22_gixsql_cobol_freebsd/

Krasner, Dima. lazy-utils, including cttyhack. BSD 2-Clause license. https://github.com/dimkr/lazy-utils/

Coughlan, Michael. Beginning COBOL for Programmers. Apress. ISBN-13 (pbk): 978-1-4302-6253-4. ISBN-13 (electronic): 978-1-4302-6254-1. Practical COBOL programming reference consulted while building CrumbStation.

Conversations and Personal Recollections

Private conversations and correspondence with people familiar with IBM systems, midrange history, MAPPER, DDM, and related computing environments.

Personal recollections from working with AS/400 support, IBM Rochester, commercial Linux distributions, and later infrastructure work. These are used as personal context, not as independent historical proof.

Note on Sources

Wikipedia was used mostly as an orientation tool: a way to find names, dates, adjacent systems, terminology, and paths for further digging. It helped trace the lineage backwards and sideways, but the main argument of this essay relies wherever possible on manuals, IBM papers, product literature, archive material, books consulted, implementation notes, and personal recollection.

In other words, Wikipedia was useful as a map. The journey itself was made through the manuals, books, and systems.