Continuing to Profit

from

Legacy Assembler Code

SHARE 100 (Feb. 2003), Session 8132

February 27, 2003

John R. Ehrman
ehrman@us.ibm.com or ehrman@vnet.ibm.com

International Business Machines Corporation
Silicon Valley (nee Santa Teresa) Laboratory
555 Bailey Avenue
San Jose, California 95141

Synopsis:

Your organization may have made substantial investments in code written in Assembler Language, and might be worried about how to maintain, upgrade, convert, or even discard and replace it. This presentation describes possible scenarios for managing legacy Assembler Language code, discusses cost/benefit trade-offs among them, and shows how you can modernize Assembler Language applications to minimize maintenance costs while maximizing returns on your organization's long-term investments.

Disclaimer

Information about companies and products mentioned here is provided solely for your reference. They are not certified or recommended.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Copyright Notices and Trademarks

The following terms, denoted by an asterisk (*) in this publication, are trademarks or registered trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM	ESA	System/370	System/370/390
System/390	MVS/ESA	OS/390	VM/ESA
VSE/ESA	VSE	z/OS	z/VM
z/Architecture	zSeries		

References

References are indicated in the text in square brackets, as in [1]. The sources of the referenced information can be found beginning on page 57.

Acknowledgments

The following people provided helpful information and references for which I am grateful. Naturally, they are not responsible for any possible misunderstanding, misquotation, misinterpretation, or misrepresentation on my part.

- Helpful observations were provided (by personal communication, or on the ASSEMBLER-LIST [36] discussion group) by: Avri Adleman, Steve Comstock, Peter Farley, Mark Hammack, Tom Harper, Andy Mau, Chris McGarvey, Walter Noniewicz, Mike Stack, Martin Treubner, and Janice Winchell.
- David Alcock's web site provides pointers to other sites describing conversion tools. (See “Web Sites” item 9.)
- Abe Kornelis's web site discusses Assembler Language programming. (See “Web Sites” item 10.)
- Helpful reviews were provided by John Ganci, John Kapernick, John Leake, Chris McGarvey, Tom Ross, Lee Wilcox, and Janice Winchell.

Corrections and suggestions for improvements are welcomed.

Contents

Tables

Executive summary

If you feel a need to “do something” about a legacy Assembler Language application, you might consider these options:

1. Keep the current application and modify it as needed.
2. Discard the application, and replace it with a purchased vendor package.
   - Specifying and negotiating requirements will be quite time-consuming.
   - The old system will need to be stabilized during parallel testing, implying possible lost-opportunity costs.
   - Vendor education and requested enhancements will be added expenses.
3. Use an automated conversion product to convert the assembler code to a high level language (HLL).
   - This approach is strongly discouraged; conversion tools are not very robust. The resulting code will be more of a headache than the original, and will cost more to support and run (among other reasons).
4. Understand and document the Assembler Language application, and then rewrite it in a HLL.
   - The documentation will be valuable in its own right.
   - Some Assembler Language functions may not map to the HLL; expect the HLL code to be slower and fatter.
   - Be prepared to have to “freeze” both code and data for the time it takes to get the new code into production. This may mean losing business opportunities.
   - Expect your staff to have to understand both Assembler Language and the new language for an extended period, in order to validate the rewritten applications.
5. Modernize and maintain the existing application.
   - The same understanding and documentation effort needed for rewriting can be put to good use in modernizing the application.
   - Many useful tools now exist that can help with this process:
     - The High Level Assembler has been enhanced to allow you to write much clearer and more comprehensible programs than was possible with earlier assemblers, and the High Level Assembler Toolkit Feature provides six components that greatly simplify application analysis and understanding.
   - No business opportunity need be lost during modernization.

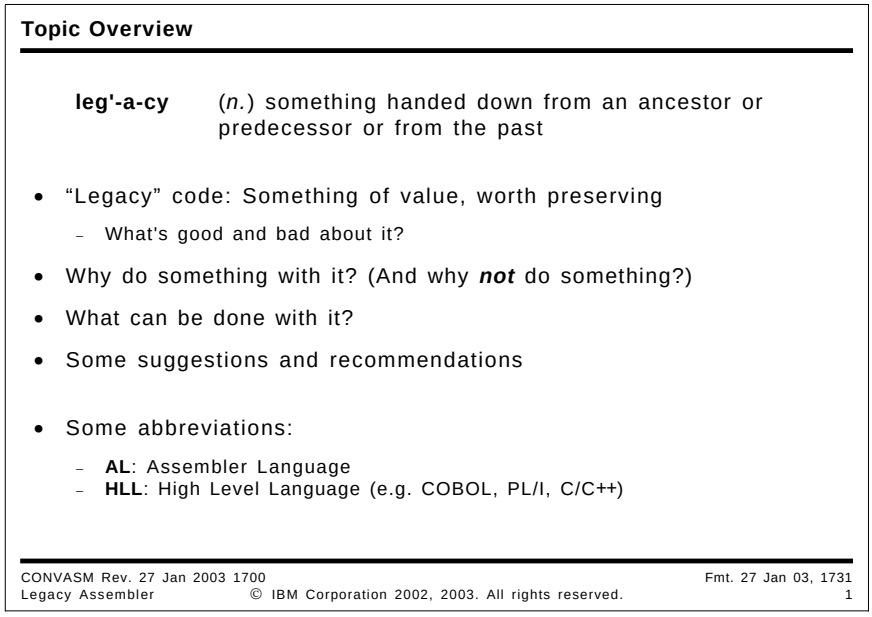
   This is the recommended option.

Recent research has shown that

- changing languages has many hidden costs, and should be avoided;
- problem-domain expertise is often more important than programming-language experience;
- high level languages do not provide improved reliability or maintainability.

Details are provided in the following pages. A compact summary of factors to consider for each scenario is provided in Table 1 on page 54.

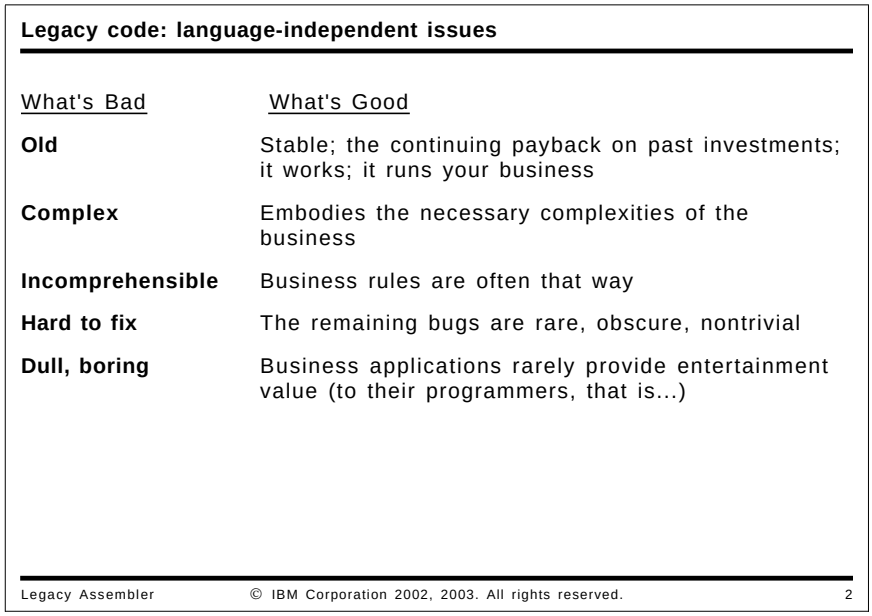
What can be done about legacy assembler code?



"Legacy" applications represent substantial investments of time, effort, and organizational culture. In rapidly-evolving business environments, these applications may require regular maintenance and upgrades.

We will investigate some characteristics of legacy applications, some typical motivations for wanting to "do something" with them, and several approaches to "doing something". Finally, we will summarize a set of recommendations and add some observations that may help with understanding the many factors involved with managing legacy code.

While our observations are intended for an IBM "mainframe" audience, most of these findings apply to legacy code written in any language. (As Sneed noted, "Of the largest 2000 enterprises in the world, over 90% employ an IBM-390 as a central server.") [12]



Legacy applications have several facets: while they often embody the complex rules and procedures characterizing the ongoing business activities of an organization, they are also expected to adapt easily to rapid changes in the business environment. There is no easy solution to the often conflicting requirements for stability and adaptability.

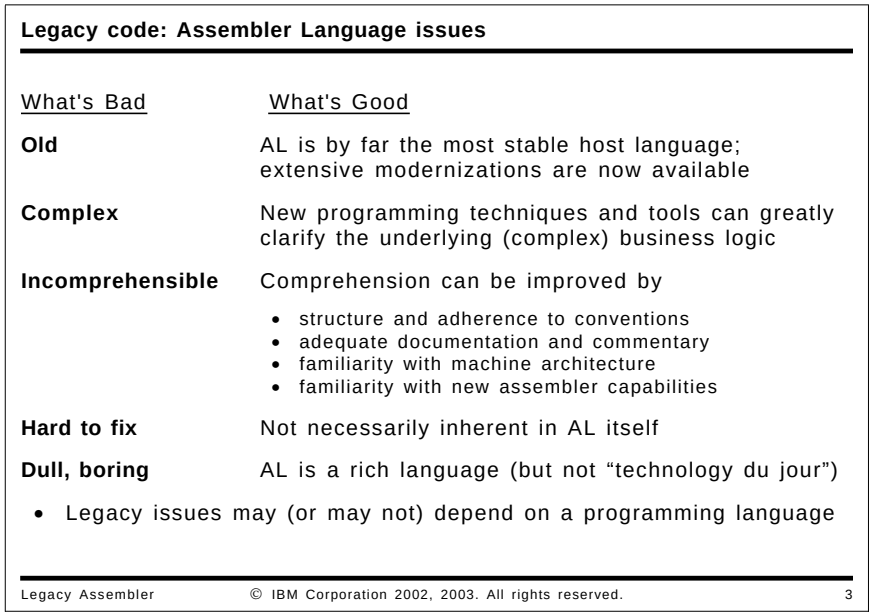
Because many legacy applications are written in languages now considered “old-fashioned”, organizations may feel impelled to modernize them somehow. This impulse may be intensified when the applications were written in Assembler Language. However unfashionable it may seem, Assembler Language is still used for major application suites, especially where code size and performance are important.

Before starting any conversion attempt, you must address two key considerations:

- Understand what the application does (and why, and how it does its job). Otherwise the converted application will be less comprehensible than the original.
- Assess the quality of the code, because it will have a major impact on what you can do with it. (It might be more profitable simply to “clean up” the program and not convert it.)

The use and value of the application to your organization are usually more important considerations than the programming language in which it is written. If the application runs often or for long periods (or both), converting it from Assembler Language to a high-level language will probably mean that hardware capacity must be increased. Other possible problems with conversion are detailed in the following pages.

We will see that the language used to program an application has little effect on code quality and maintainability, nor on programmer productivity. Frederick Brooks [1] says “*I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.* We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems. ... If this is true, building software will always be hard. There is inherently no silver bullet.”

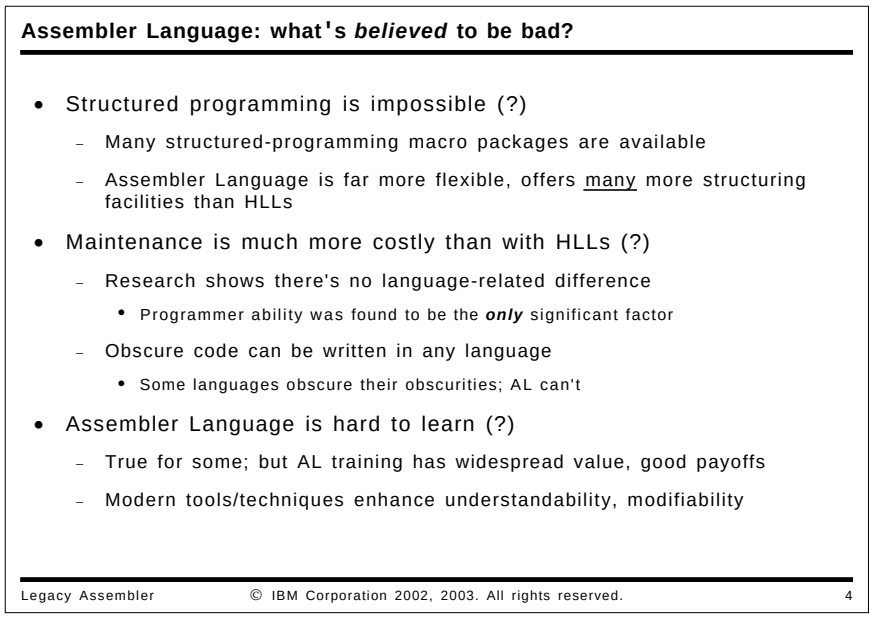


Some legacy-application concerns focus specifically on Assembler Language. The language is indeed the oldest of all surviving programming technologies; but on any platform, it is also the most stable. (Bear in mind that programming in a HLL means trusting a compiler and its run-time library to be correct. With Assembler Language, there is no intermediary: “What You Write Is What You Get”.) We will see that many of the problems and difficulties associated with Assembler Language applications are typical of all software, and are now more easily managed.

The complexity of an application program is a mixture of the necessary complexity of the business logic it implements, and the quality of the encoding of that logic in a programming language. A lack of discipline in the use of any programming language can make it harder to grasp the intent of the business logic it embodies.

Assembler Language code written many years ago may have suffered from the lack of modern facilities for managing such details as program structure, register management, data structuring, and more. Modern assembler tools and technologies can help in many ways: the High Level Assembler supports many new language extensions, cross-references, and external interfaces; many tools provide debugging, source library scanning, graphical program understanding, and other facilities to simplify application development and support.

Existing application code can be modernized — restructured, documented, and generally improved — using these technologies. We will review these new techniques in “High Level Assembler” on page 37.



Assembler Language has some undeserved negative associations, such as:

- Structured programming is impossible (?)

  This is in fact not true. Assembler Language provides a flexible repertoire of techniques for structuring multi-module applications in many different ways. Within a single module, traditional structured programming statements (like IF-THEN-ELSE, DO-WHILE, etc.) are easily implementable using available structured-programming macro packages. (One of these is described in “High Level Assembler Toolkit Feature” on page 40.)

- Maintenance is much more costly than with HLLs (?)

  This is in fact not true. Recent research [11,15] has shown that the application's programming language has no significant effect on application maintainability.

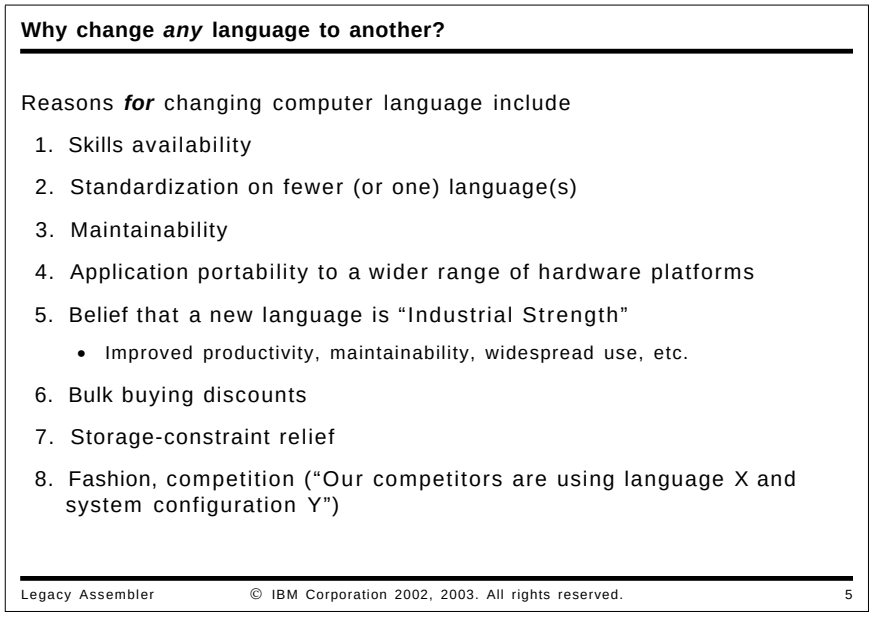
- Assembler Language is hard to learn (?)

  Assembler Language involves understanding two languages simultaneously: the “language” of the machine (its instructions, registers, data formats, etc.), and the “language” presented to the assembler to be translated into the object code of the machine's language.

  People new to Assembler Language may consider it old-fashioned, but understanding what's happening “under the covers” of any HLL is important to all professionals. Education providing a basic grasp of Assembler Language (and therefore, of machine language) provides valuable benefits even to programmers who work mainly with HLLs.

Further discussion of Assembler Language as an implementation language is provided at “Assembler Language: pros and cons” on page 51, and at “Web Sites” item 10.

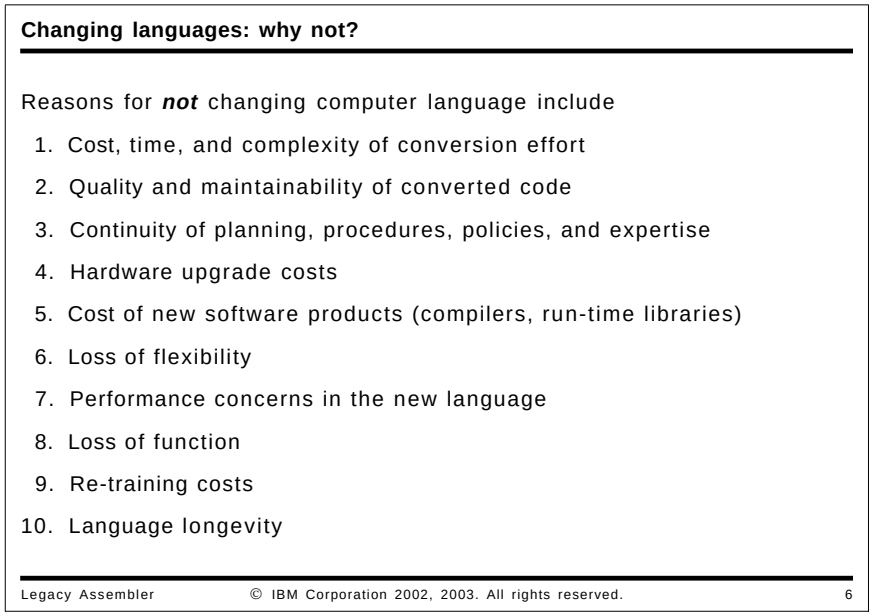
Why (and why not) consider changing languages?



Among the reasons an organization may wish to undertake a programming language conversion are:

1. **Skills availability**. Younger programmers are often not trained in "legacy" languages, and therefore may not have the skills needed to maintain existing applications.
2. **Standardization** on one language. "Organizations had found a proliferation of languages in use.... The result was inflexibility in the allocation of technical staff and higher support costs. Therefore a new language was sought which would be comprehensive enough to supersede all the existing languages." [10]
3. **Maintainability**. Existing applications often appear to be too complex to understand and maintain; it is expected that migrating to a different language will improve understandability and therefore maintainability.
4. **Range of hardware platforms**. Organizations sometimes must support a variety of machine and operating system types, and therefore intend that the "...language chosen should available on a wide range of computers. This was necessary to avoid 'proprietary lock in' which would raise prices on hardware purchase and maintenance." [10] Applications written in HLLs are *usually* more easily ported to other platforms.
5. **Industrial strength** of the new language. "...the language selected should be able to tackle all the tasks being handled by the existing languages. This indicated the need for a large general purpose language." [10] The expected benefits include structured programming, widespread use, increased development productivity, perceived maintainability, and availability of staff.
6. **Bulk buying discounts**. "Replacing the fragmented approach to the purchase and training on assorted computer languages with a unified approach was anticipated to be cheaper." [10]
7. **Storage-constraint relief**. Older programs may have been written at a time when central storage was limited, so they have difficulty handling today's larger volumes of code and data in the area below the "16MB line".
8. **Fashion, Competition**. The software industry is frequently driven by fads and fashions that have little to do with real business requirements. Sometimes language conversions are undertaken because it appears that competitive organizations have done so.

We will revisit these points in our summary observations on page 41.



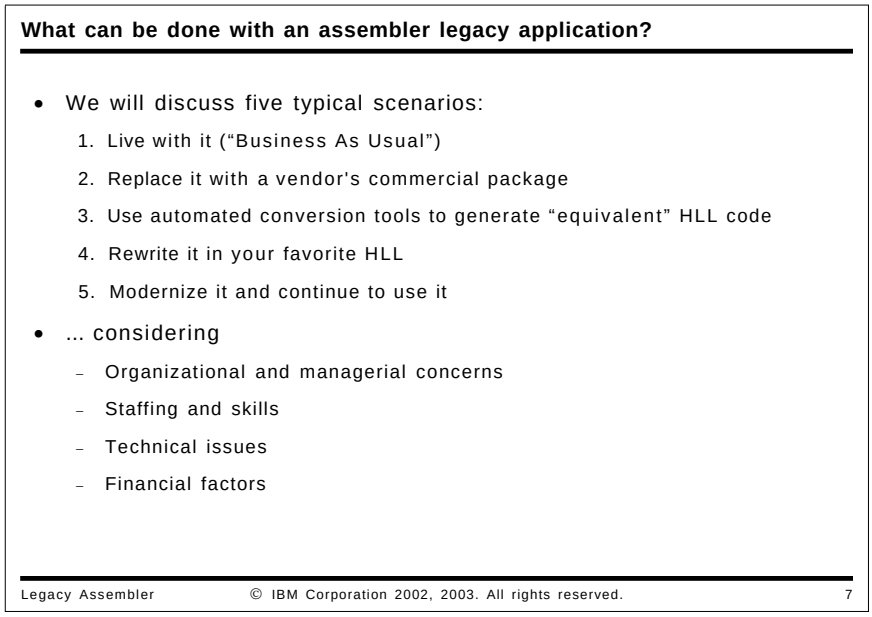
Among reasons given not to undertake a programming language change are:

1. **Complexity of conversion**. The conversion task may appear too large to be worth doing.
2. **Quality of converted code**. There appear to be no easy ways to get converted code as good as the original.
3. **Continuity** of procedures, processes, and expertise. Too much disruption of current practice might occur.
4. **Hardware costs**. Organizations don't want to be forced to upgrade system capacity due to less efficient applications.
5. **Software costs**. New compilers, run-times, and support tools may be more expensive.
6. **Loss of flexibility.** A new language might offer less control over code and data layouts.
7. **Performance.** Equivalent (or acceptable) performance may be difficult to achieve.
8. **Loss of function.** These concerns include access to specific machine instructions and operating system services, as well as excess generality and complexity of a high-level language.
9. **Re-training costs.** Existing staff may require re-training in the new language.
10. **Language longevity.** Not every programming language has had a long lifetime.

We will expand on these points in “Summary observations” on page 41.

Any extensive transition is difficult. A change of programming language is only one factor in converting or migrating an application. Consider Holmes' comment: “Paul Strassmann, at one time the Pentagon's information chief, has observed eight ‘build-and-scrap’ cycles in computing investment since 1946, with each cycle greatly surpassing the previous one both in absolute value and percentage of overall business investment. Strassmann projects the next binge to be two and one-half times more expensive than the last....” [4]

What you might do with legacy code: five scenarios



We consider five typical responses to the question "What should we do with our legacy code?"

1. Continue using and updating the existing application. ("Scenario 1: Live with the current application" on page 9)
2. Replace the application with a purchased or leased "package". ("Scenario 2: Replace the application with a vendor package" on page 12)
3. Convert the application to a new "target" language using an automated conversion tool. ("Scenario 3: Convert to a HLL using automated tools" on page 14)
4. Rewrite the application in a new "target" language. ("Scenario 4: Rewrite the application in a HLL" on page 20)
5. Modernize the application and continue to use it in its original language. ("Scenario 5: Modernize and maintain the application" on page 35)

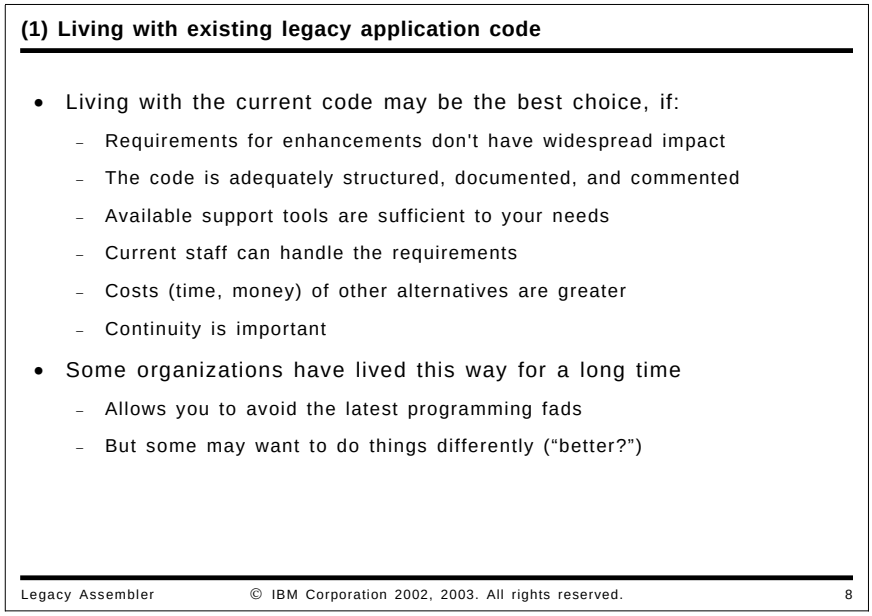
Other responses are possible, of course; but these five provide considerable insight into the results we might expect.

In discussing these options, we will also consider their possible effects on

- organizational and management concerns
- staffing and staff skills
- technical issues
- financial factors.

While much of the following discussion applies to legacy code written in any language, we will emphasize factors that apply to Assembler Language.

Scenario 1: Live with the current application

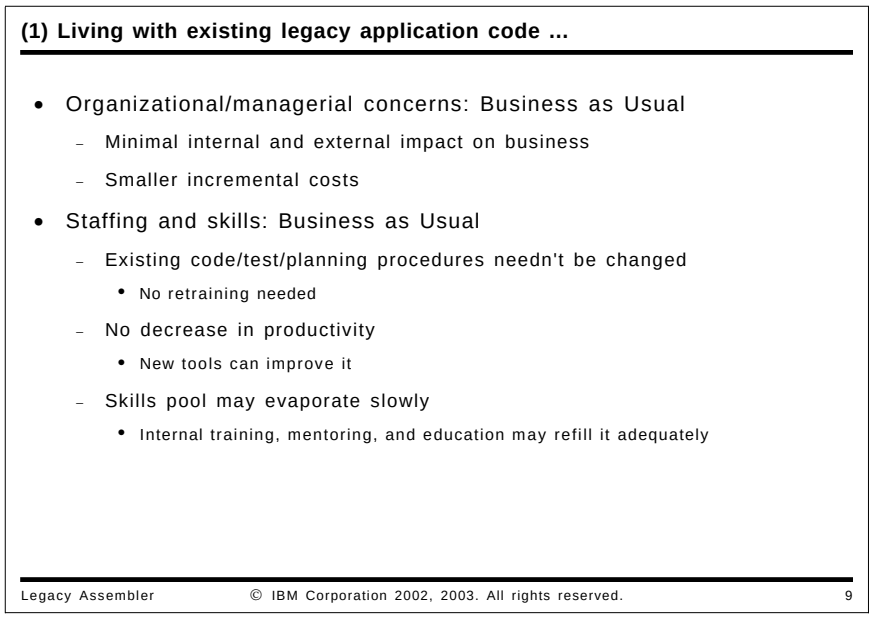


Because other alternatives are usually expensive and disruptive, you may want to somehow live with what you have. Martin Ward [16] observes, “For many of these so called *legacy systems* the option of throwing the system away and re-writing it from scratch is not economically viable.”

Indeed, there may be no strong motivations to change; Harry Sneed noted that “Recent studies show that 80% of the actual IT-production is carried out by legacy systems.” [12] Businesses may be understandably reluctant to make major changes.

Continuing to use the existing application can be the best choice if demands for change aren't severe: its incremental costs are smallest. You may decide to “live with” the existing application if:

- requirements for enhancements can be handled by current staff using current processes, procedures, and tools;
- the application code is sufficiently readable that updates can be made reasonably easily;
- the existing support staff understands the application and the business processes it supports;
- doing things differently will create higher costs and pain levels.



Among the benefits of “living with” the current application code are:

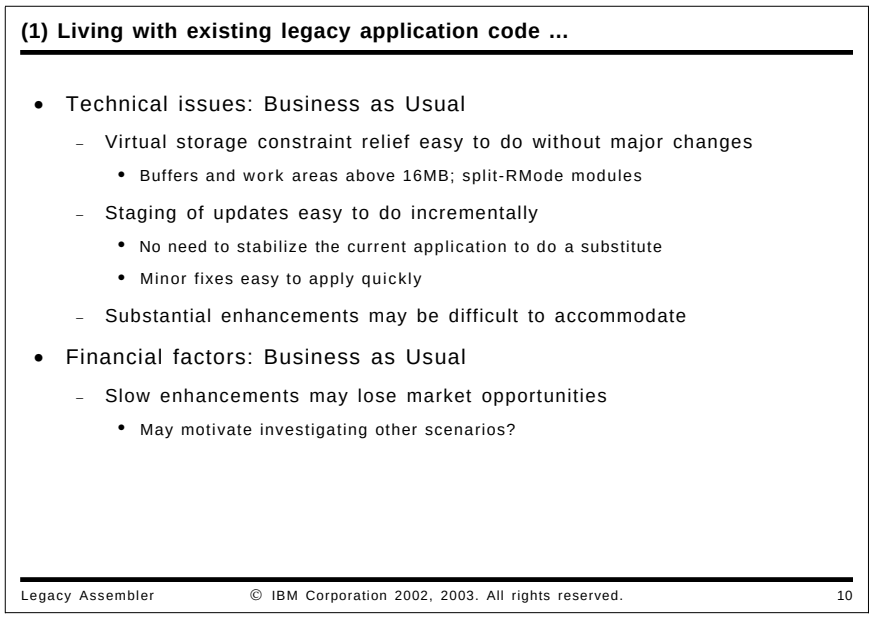
- Disruptions of existing management processes are minimized.
- Incremental costs are smaller than with other choices.
- Current staff does not need re-education in language, planning and estimating skills, coding and testing procedures, and product shipment and installation techniques. Their knowledge and experience can be used to help train newer staff.

  An additional benefit is that there is no decrease in productivity, while new tools can help improve it.
- Application updates can be made as needed and when they're ready; changing to a new system may require stabilizing the old and running old and new in parallel for an extended period.
- 31-bit enablement (using storage above the “16MB line”) is usually straightforward:
  - Virtual storage constraints — particularly, limited storage below the 16MB “line” — are relatively easy to relieve. I/O buffers and data work areas may be moved above 16MB using simple and straightforward procedures, without major modifications to the overall structure of the application.

    McGarvey's SHARE presentation [35] provides an excellent overview of such techniques.
  - Similar considerations apply to large “load module” executables. Previously, any component of an application module that required residence below 16MB forced the entire module to reside there. The z/OS Program Management Binder supports “split-RMode” PDSE modules in which only those elements that must reside below 16MB are loaded there, while elements that can reside “anywhere” are loaded above 16MB.

Recent High Level Assembler enhancements and support tools can simplify programming and maintenance tasks; we will briefly describe these in “High Level Assembler” on page 37.

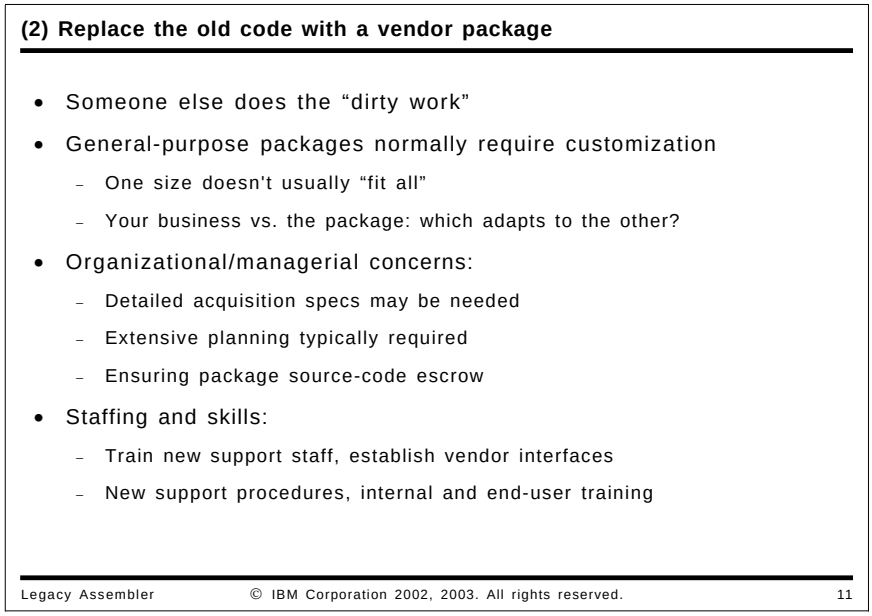
Despite claims implying that “modernization” is a business requirement, this “live with it” approach may allow resources that might otherwise be spent on new languages or techniques to be focused on immediate business concerns. We will see later that this scenario has many similarities to “Scenario 5: Modernize and maintain the application” on page 35.



Some possible disadvantages of this scenario are:

- Staff familiar with the application and its language may be harder to hire and retain (but internal training and education may help).
- Major upgrades may be difficult for various reasons, and an inability to make such upgrades rapidly may cause business opportunities to be lost.

Scenario 2: Replace the application with a vendor package



In discussing the “Buy versus build” question, Brooks [1] comments “The most radical possible solution for constructing software is not to construct it at all....”

The option of replacing a “home-grown” application with a commercially available vendor package has several attractions, not least of which is that management may believe they need no longer deal with what may seem a difficult and unresponsive programming staff.

However: vendors naturally desire the broadest possible market so that they can capture as many customers as possible. Their application packages therefore tend to be very general, and require customization and tailoring for each individual customer. As Brooks observes, “Much of present-day software-acquisition procedure rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software-acquisition problems spring from that fallacy.” [1]

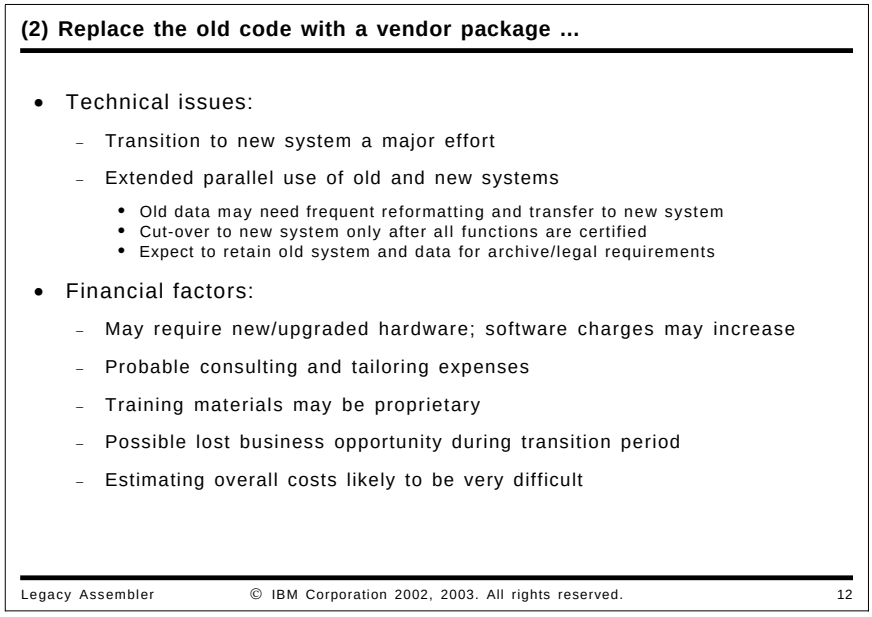
Because the legacy applications that run your business will almost always be different from the capabilities of a vendor package, you will have to decide how much your business will change to accommodate the package, and how much the package may need tailoring to fit your business processes and requirements. The costs of these adaptations may be extensive and difficult to estimate, especially as they will involve changes of employee habits.

Management also must prepare budgets and detailed specifications for future requested implementation of local requirements, and must expect that long-range planning is needed for the period of time when the new application is being tested and tuned while the old application continues to work.

Leased or purchased software packages may become important to your organization, so you should prepare for the possibility that the vendor may go out of business or be acquired by another company with no interest in the application. Take care that any vendor contract includes provisions for access to the package's source code in case the vendor can no longer support it.

Your programming support staff will likely need to be trained to provide interfaces to the vendor for problem resolution and requirement submission (or to make modifications to the package

locally), and training and support procedures must be provided for the end users of the application who must learn the new product.



Technical staff must prepare for an extensive period of parallel execution of old and new applications, to verify that all functions are properly supported. This may impose an additional burden on people who are already fully occupied; additional knowledgable staff may be required from the vendor (for a price). The vendor may also impose substantial consulting, tailoring, and training fees.

Once the new system is in place, existing data will have to be reformatted and transferred into the new system, possibly on a regular basis. Also, the old system and its data will probably have to be kept available for an extended period, in order to satisfy record-retention requirements.

Financial considerations of this choice include:

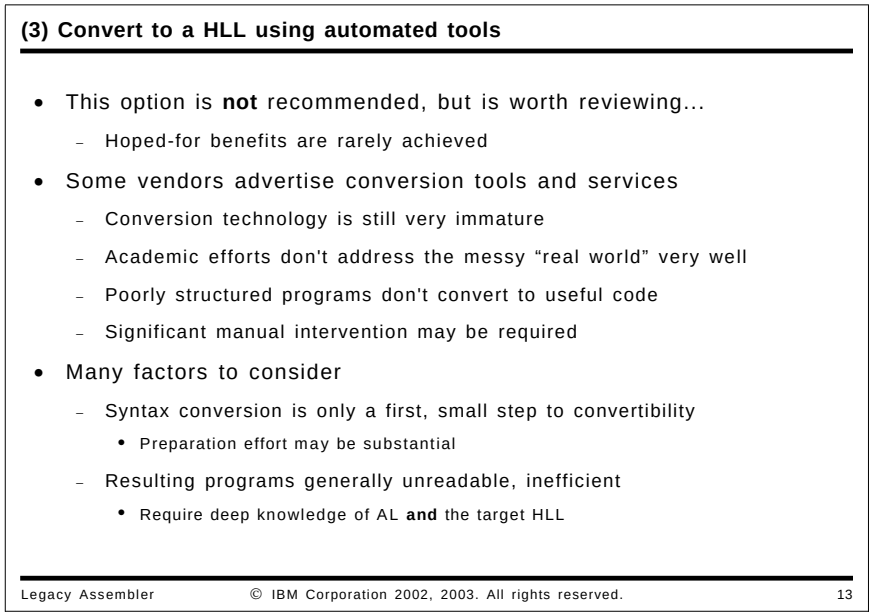
- New or upgraded hardware may be required if the vendor package does not perform as well as the retired application.
- Installation and customization typically require vendor consultants. Additionally, training materials for the new package are usually proprietary, so the vendor will impose additional charges for courses.
  - Upgrades and enhancements to the vendor package will require additional and proprietary training. “The training costs were higher than anticipated because courses were proprietary and subject to limited competition. The suppliers also released frequent upgrades and additions that required extra training. Apart from the direct cost of training this caused disruption.” [10]
- Software charges for the vendor package must include the costs of its support and maintenance.
- During the period of parallel execution of the old application and the vendor package, it is unlikely that any enhancements can be made to either without risking serious disruptions. This loss of business opportunity may or may not be significant.

In general, it is quite difficult to create accurate estimates of overall costs: people, consulting, software, hardware, and opportunity.

Scenario 3: Convert to a HLL using automated tools

Converting applications from one language to another is not simple, and the process is quite difficult to automate satisfactorily.

Many of the factors involved in rewriting an Assembler Language program in a HLL (as described in “Scenario 4: Rewrite the application in a HLL” on page 20) also apply to preparing for automated conversion. However, because the results of automatic conversions are generally unsatisfactory, we will wait to discuss those general conversion considerations in detail there.



For any large application, it is natural to hope that language conversion can be automated. However, even what appear to be the best available tools have serious limitations. Do not expect easy, accurate, reliable, or usable conversions.

- “Many software researchers advocate rather than investigate. As a result, (a) some advocated concepts are worth less than their advocates believe, and (b) there is a shortage of evaluative research to help determine the actual value of new tools and techniques.” [5]
- “Recently several companies have started advertising translation systems of various kinds, including assembler-to-C. From the available information it seems that these tools are... brute-force literal translator[s], with little or no further analysis.” [3]

  “...the literal translation approach, besides being grossly inefficient... fails to recognize idioms that are architecture-specific and would not have the same meaning when translated literally.” [3]

A “brute force” conversion simply maps the original Assembler Language program to a HLL in which registers and storage are mapped into HLL variables that mimic the original machine's architecture and instructions. These programs are highly unsatisfactory, as they lose none of the apparent complexity of the original program, while adding obscurity.

Feldman and Friedman [3] note that brute-force translation results in code that is typically 10 times slower; it is often easier to debug (because it looks like the original!); but it is difficult to modify and maintain. Only two published studies [3,16] seem to have approached the problem with any generality; we will refer occasionally to their experiences (which aren't encouraging).

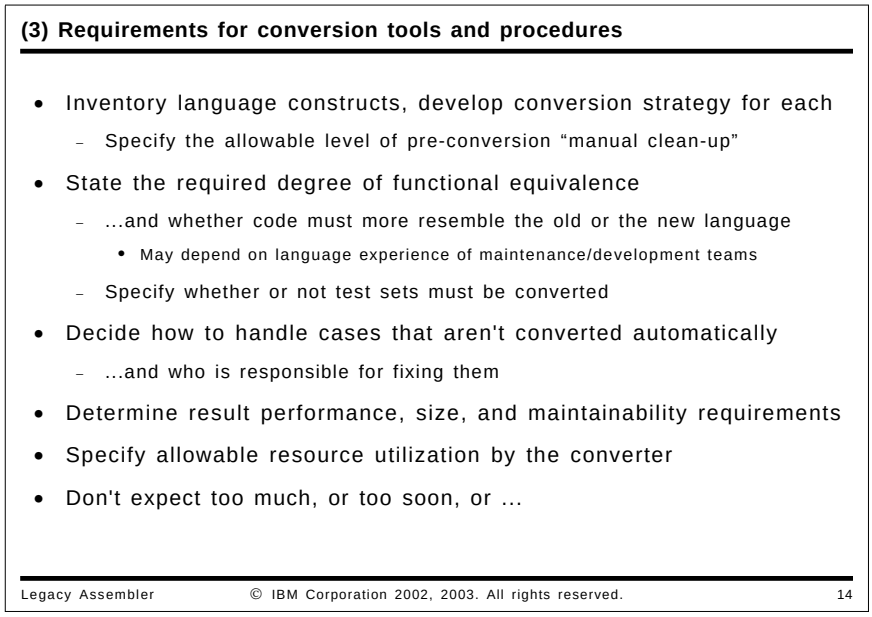
Thus, research results in the area of language conversion have focused mainly on attempts to capture something of the structure of the original program. Several authors (including the researchers themselves!) have commented on the relatively low quality of the results:

- The conversion process is rarely fully automatic: Cristina Cifuentes comments “Companies that provide a translation service will make use of human resources to aid in this area.” [2]
- Ward [16] describes a substantial effort to create a conversion engine “... which models as accurately as possible the behaviour of the original assembler module; without worrying too much about the size, efficiency, or complexity of the resulting code.” (This view may not precisely describe your requirements.)
- “Conversion technology... is still in its infancy. ... automated language conversion is much more difficult than many people anticipate. ... language conversions are grossly underestimated — even by well-known reverse-engineering experts. ... Harry Sneed summarizes the state of the practice in the transformation marketplace as follows: ‘The reality looks different. Those who can read between the lines recognize that the problems are grossly simplified and that the advertised products are far from being ripe for use in practice.’” [13]
- “...research is carried out in an academic setting with a vague expectation that it will later be transferred to industry, an expectation that is rarely fulfilled.” In studying other tools, Feldman and Friedman (who have achieved better results than most) comment: [3]
  - “...was not applied to real-world programs”
  - “...such tools are not automatic, and need manual supervision”
  - “This tool is not automatic, it is a convenient environment to aid a human programmer”
- “Due to automation problems, most conversion tools apply the technology of syntactic conversion. Even with this apparently simple and low-level approach, many difficulties occur, and the scale of those intricacies is not yet fully understood.” [13]
- “...the CISC techniques had only been tested with small, unoptimized code, and hence were not exposed to very large unstructured examples which make the restructuring process harder to achieve.” [2]

The feasibility and cost-effectiveness of legacy-software re-engineering remains a theme for debate. Do not expect high levels of abstraction or comprehensibility in converted programs.

The technical difficulties of code conversion, while substantial, can be dwarfed by the problems of capturing the business rules the code implements. While well-commented code can partially explain its actions and purposes, understanding the reasons for each step of the program may be difficult to reconstruct after conversion.

Specifying requirements for conversion tools



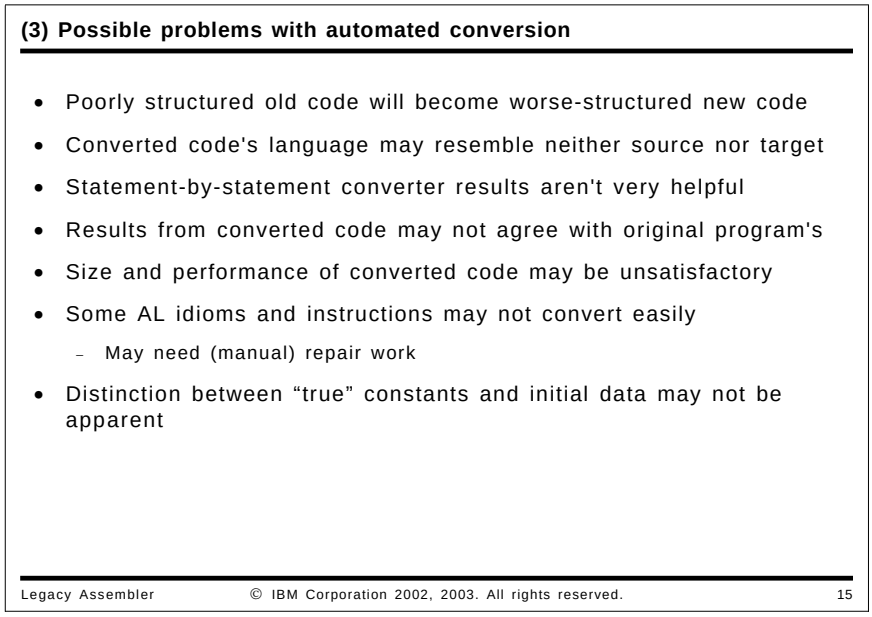
If you are seriously considering using an automated conversion tool or service, Terekhov and Verhoef [13] offer these *minimum* recommendations for a source-to-source converter:

- “You must inventory the native and simulated language constructs of the system that needs conversion.”
- “You must develop a conversion strategy for each language construct. Specifically, you must list the input and output fragments that describe the converter's desired behavior.”
- “You must make an explicit statement about whether the converted system should be functionally equivalent to the original system. Intuitively, you would think that this is always so, but in practice, a sophisticated automated modification effort usually exposes faults and unsafe code in the original system. Often, the customer then requires the developers to fix these faults as well. Thus, potential requirements creep must be dealt with in the requirements. Note that it also hampers testing the new system because regression testing is based on equivalence.”
- “You must include a statement about whether the original system's test sets are to be converted. If errors in the tests are exposed, the requirements must state the policy toward modification.”
- “You must set as your goal maximum automation of the conversion (enabling minimal human interference).”
- “If you plan to maintain the converted system, you must make it maintainable. For instance, if the original maintenance team is going to continue in its role, the new system should be as similar as possible to the original system so that the team can recognize the original code. If, on the other hand, the maintenance team is new, the conversion should try to use the target language's idiom so that maintainers recognize the code as normal for that language.”
- “You must make the converted system's efficiency adequate both in compilation and execution time.”
- “If you will use the converter many times, the time required for conversion is relevant. It is not always feasible to optimize this without distributing calculations over many machines.”
- “If you plan to maintain the converted system, its size must not exceed the original system's size by much (if at all).”

The authors conclude these recommended steps with these observations:

- "Requirement specifications usually mention only the native-to-native part of the [conversion] mapping, in the form of a statement-by-statement conversion. This phenomenon is in accordance with people's tendency to focus first on the easiest problems. One of us (Chris Verhoef) was an external reviewer of several large-scale conversion projects. Most of them failed because the hard problems were avoided in the requirements."
- "Apart from these explicit requirements there is always an implicit understanding of how the converter will work. This reflects the *customer's expectations of the advantages associated with transferring the system to a more contemporary environment. These imaginary advantages often motivate the conversion process but they are rarely realized. A popular misconception is that after conversion the system is change-enabled so that totally new features can be implemented easily.* (Emphasis added) The problem is aggravated by companies marketing conversion software as yet another silver bullet; the quality of such converters is often less than optimal, and in some cases even nonexistent."

Thus, you should be prepared to establish strict and detailed requirements for what you hope to get from conversion tools (and not set your expectations too high).



A variety of problems may be encountered with the results of an automated conversion process:

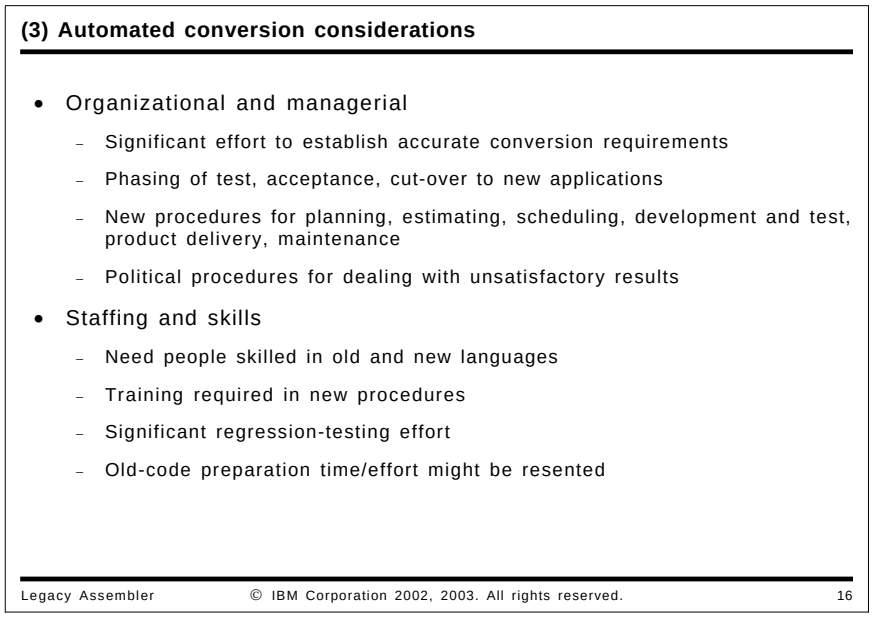
- It is unreasonable to expect a tool to convert programs that are poorly structured, even though this may be one of the reasons for attempting a conversion in the first place. Feldman and Friedman observed [3] that "Our experience with Sapiens code has shown that large and complex assembly-language programs that are maintained by several different programmers contain patches and unnecessary code...."
- The code created by the converter is very unlikely to produce the same results as the original program.

  One conversion technology that claimed considerable success noted "We did not, however, manually check the semantics of each generated file against the original assembler!" [16] You might reasonably require that the converted program produce the same results as the original, and expect that the results would not require manual checking.

  Much of this additional effort is needed because converted code does not capture the "algorithmic" content of the original program; most conversion tools can capture only "microstructure", at best.

- Performance of converted code is generally much slower, and code sizes much larger, than the original program's. One of the most successful conversion efforts observed [3] “The translated C code was only three times slower than the original, which in our opinion is quite reasonable for such translation. In fact, it is less than 10% slower than the hand-crafted C version on the same platform.” (An indirect acknowledgment of generally superior performance from Assembler Language code!)
- While conversion tools may claim generality, there are many cases where conversion may be incomplete or inaccurate. Feldman and Friedman [3] observed “A large part of the development of [the converter] was dedicated to many theoretically-unimportant details, such as supporting most of the IBM 370 instructions.” Converting any Assembler Language application will necessarily involve such details.

  Some Assembler Language constructs may not have equivalents in the target language. A conversion engine should indicate which parts of the program could not be converted. For example, one conversion tool recommends “In addition, the user needs to check for FIXME comments in the generated C code which indicate areas where the translated code may be incorrect...”. [16]
- A converter may not make the important distinction between literals (“constants”) and variables, “thus losing the important information of their immutability...”. [3]



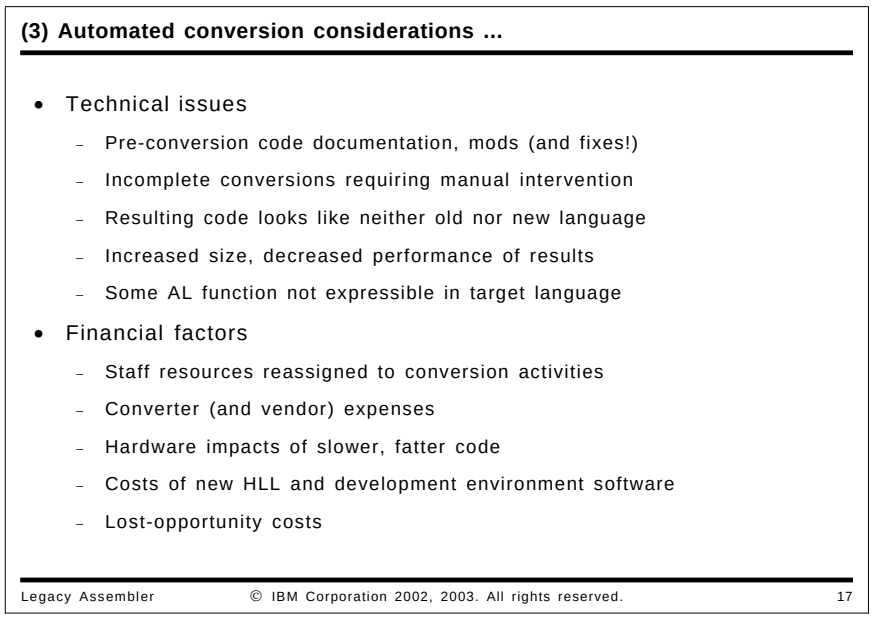
According to Feldman and Friedman, [3] “There are two distinct modes of working with automatic translation. It can be used as a one-time effort to translate a large piece of code, which will then be debugged and maintained independently. This requires the target code to be readable and properly documented. ... [the other] is having two working versions of the same product. ... the development and maintenance will be done on the assembly-language source, and nothing will be done on the target code (except for verifying that it works).” Most people will adopt the first of these modes, as the second doesn't replace the original application.

Planning for automated conversion will require a substantial investment in setting detailed requirements for conversion activities. Otherwise, the conversion effort will be repeated many times as each difficulty is discovered.

Assuming the conversion progresses satisfactorily, scheduling plans must include regression and acceptance testing and the eventual replacement of the old application by the new. The costs of this validation may be quite significant. New procedures must accommodate the changes to all affected aspects of development and maintenance; planning and estimating must account for dif-

ferent skill sets; development, test, delivery, and maintenance processes may change; and customer support procedures may need updating and training.

During the transition period, heavy demands will be made on individuals who know both the original and target languages, as they will be required to validate the conversion. They will also be called on to restructure the old code to make it more acceptable to the converter; Feldman and Friedman [3] found that the staff responsible for "cleaning up" the old code were unhappy to be doing what they perceived to be unproductive "busy work". Those staff who know only the new target language will have to be trained in the new development and maintenance procedures, and will probably be heavily involved in testing.



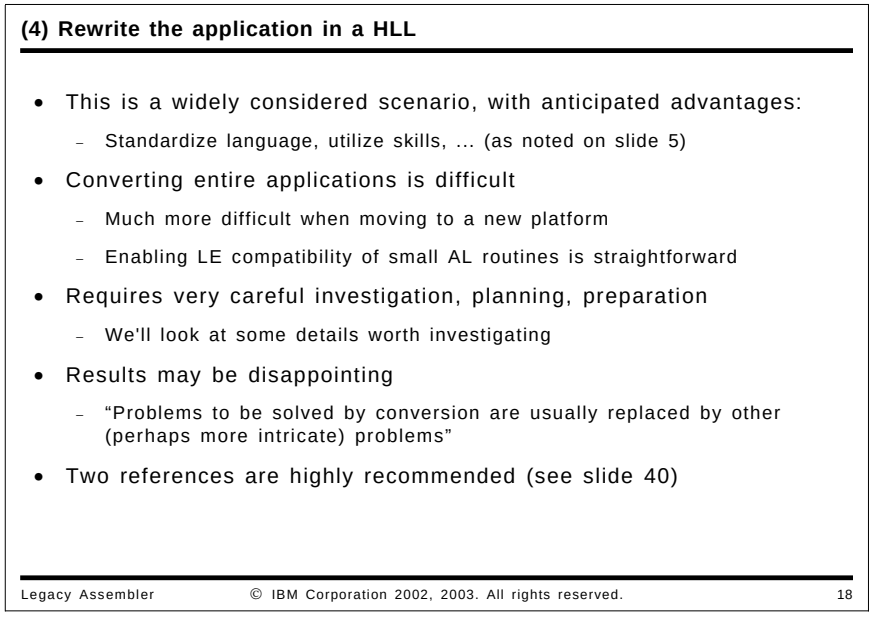
Preparing for conversion often exposes problems in the application. These must be fixed and tested before conversion is attempted. Converted code is frequently incomplete, and may need manual repair and extension before it is usable; this is even more common when the original code is written in Assembler Language, because some instructions and idioms have no HLL counterpart.

Code converted from Assembler Language to a HLL is almost always slower and fatter than the original, which may impose additional constraints on the converted application. The resulting code can be very difficult to understand, as it will probably resemble neither the original Assembler Language nor a "normal" program written in the new HLL, but will appear to be a hybrid mixture of both.

Financial aspects of a conversion could include:

- the costs of reassigning staff from normal activities to the conversion effort
- expenses for use of the conversion tool, and consultation and training in its use
- possible needs for hardware upgrades due to slower and fatter code, as well as the indirect costs of procurement delays
- expenses for new language compilers and run-time libraries, as well as new development, maintenance, and support tools
- the costs of lost business opportunities during the period when the old application must be "frozen" until the new application is in productive use.

Scenario 4: Rewrite the application in a HLL



The problems of converting programs from *any* language to another is an area of active research; [10,13] converting Assembler Language code is particularly challenging. [2,3,6,8,16]

Two substantial (and highly recommended!) studies provide useful perspectives on language conversion:

- Terekhov and Verhoef [13] discuss language conversion in general;
- Middleton [10] studied 16 organizations that considered converting applications to a higher-level (4GL) language.

In both cases, the authors observe that conversions require very careful investigation, planning, and preparation, and that the results are usually disappointing.

Because Assembler Language programs are closely tied to the architecture of their hosting system, it is reasonable to assume that a converted program will run on the same system. Conversion problems are complex, even when no change is made to the hosting platform. If different hardware architecture and operating system environments are targeted, many additional (and serious) problems must be addressed.

If an entire application need not be converted — because only small portions such as callable subroutines are written in Assembler Language — then conversion to another language may not be required. Converting assembler subroutines (of HLL applications) to execute correctly in new environments (such as the OS/390 and z/OS "Language Environment", LE) is fairly straightforward, and has been documented in various IBM manuals [18,27,28] and SHARE presentations. [34]

Remember that "Any decision maker considering language or dialect conversions to solve a problem should realize that *the problems that are perceived to be solved by the conversion will be replaced by other, perhaps more intricate, problems.*" [13] (Emphasis added.)

Many observers have noted that conversion by hand — rewriting the application — requires knowledge of both languages. [36]

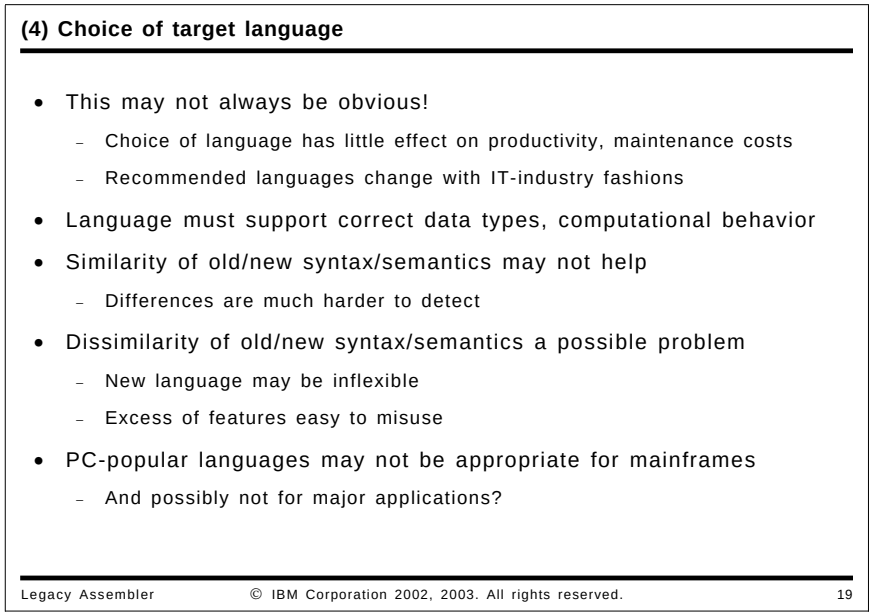
Planning and preparing for language conversion

Converting an application from one language to another (whether done automatically or manually) requires analysis of a broad range of factors. While these topics were deferred in discussing automatic conversion (Scenario 3), they apply equally well there as well as to manual rewriting (Scenario 4), to be examined here.

Topics to be discussed include:

- choice of target language
- preparing for rewriting
- understanding the application
- typical convertibility problems
- quality of the converted code.

Choice of target language



The choice of the "final" language — to which the application will be converted — is critical, as it must be able to accommodate the data types of the original language, its computational behavior, and also be understandable, modifiable, and maintainable. Unfortunately, the "latest technology" or "language du jour" changes every few years, each promising to make programming easier, faster, more reliable; this adds to the difficulty of a viable choice.

To simplify conversion, it is natural to expect that the new language should be close to the original language in syntax and semantics. "The converse of this... statement... that conversion between similar languages would be easy, is not at all true. ...conversion is *always* intricate." [13] (Emphasis in original)

Close affinity between the original and new languages may cause other difficult problems:

> ...a possible misconception that syntactic equality between language constructs of different languages or dialects would be a useful indicator of the complexity of a conversion project — that the more equal that languages are, the more easy a conversion would be. In fact, it is a very non-intuitive measure for complexity of language conversions because the more syntactically similar equal languages are, the more difficult it becomes to detect differences. In addition to all the language conversion problems we have, we must also deal with semantic differences that we cannot even detect syntactically. [13]

However, lower affinity between the two also appears to increase conversion problems. Jeffrey Voas [14] comments: "...the current craze — object-oriented languages — provides threads, polymorphism, inheritance, encapsulation, information hiding, and so on. These features cause serious problems when misused. You could argue that these complicated languages are making it harder to build quality software than if we used older languages that were less feature-rich."

Most programming languages embody a conceptual model that shapes the way you think about programming. For example, many procedural languages may manipulate data structures as normal entities, while object-oriented languages may enforce "method" interfaces to access such related pieces of data. Rewriting procedural language code in an object-oriented language may be quite difficult.

Many languages have been recommended by influential authors. To quote Middleton [10] again: "...frequent advertisements... feature slogans that relate development performance to the language used. ... There are many tasks required to construct an information system and programming is only one of them."

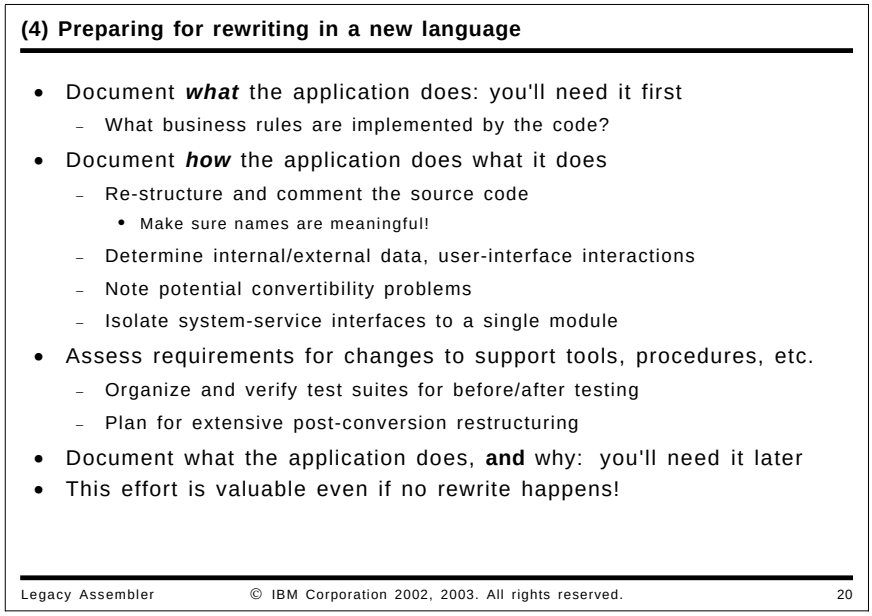
- Brooks' 1975 recommendation for a high level language was the following: "The only reasonable candidate today is PL/I."
- Boehm selects Pascal and Ada as "The strongest choice for enhancing productivity in the long run."
- Boddie recommends "Pascal, C, PL/I, or any of the Algol derivatives"....

In fact, the choice of a new language may have relatively little effect on productivity. Boehm's analysis (cited in [10]) of how programmers spend their time shows that they spend only 13% of their time writing programs. A detailed study by Prechelt found that the choice of language is not as significant a productivity factor as programmer ability. [11]

Hoped-for benefits of HLLs may be difficult to realize. Middleton [10] observes "But as languages become more specialized they become more inflexible. It is therefore apparent that a trade off is necessary between the level of a language, its merits for assisting in the production of reliable programs and its general applicability."

Languages whose value was proven on PCs may not work well for mainframe applications: on PCs, performance isn't as important; data requirements are modest; stability and recoverability aren't as critical; and user interfaces tend to be much more important than underlying function.

Preparing for rewriting in a new language



The most important step in preparing to convert an application to a new language is to understand fully what the application does, and why. This must include not only the low-level statement logic of each module, but also the functions of each internal procedure, the functions of the module itself (including the business rules it embodies), and also the interactions of the module with others in the application, and *their* interactions with user interfaces and with internal and external shared execution data. [36]

Without this detailed documentation and a solid understanding of the application's role in the business, it will be very difficult to be sure the rewritten code performs the same functions as the original. Cole adds, “Thorough documentation cannot be overstressed! Without it, it will cost more for successor programmers to understand, fix, and upgrade your code. ... Subroutine interfaces need to be thoroughly documented. Header commentary should be written to describe the subroutine's purpose, its environment, its inputs, its outputs, and its various return conditions. It should not be necessary to read the subroutine's code to determine either how to use it or what its actions and effects are upon the environment.” [29]

McKendrick reinforces this point: in studying a COBOL-to-Java conversion, he commented “The learning curve is a big issue here. The Java re-write took the longest time because the Java programmer was unfamiliar with the underlying COBOL code and business rules.” [9] Similarly, from Whittaker: “The only hope for understanding programs is good documentation of control structure blocks and detailed descriptions of the purpose and use of data structures. Moreover, effective documentation must often go beyond this detail and include design rationale and, even more important for maintainability, the reasons against alternative designs.” [17]

“Converting application software from one language to another is usually done to simplify maintenance. Therefore, the target source texts must be well structured, contain as little global data as possible, and so on. Since source language problems rarely comply with these requirements, any sensible language conversion should first start with extensive restructuring — despite the problems you will encounter with the classical restructuring tools.” [13]

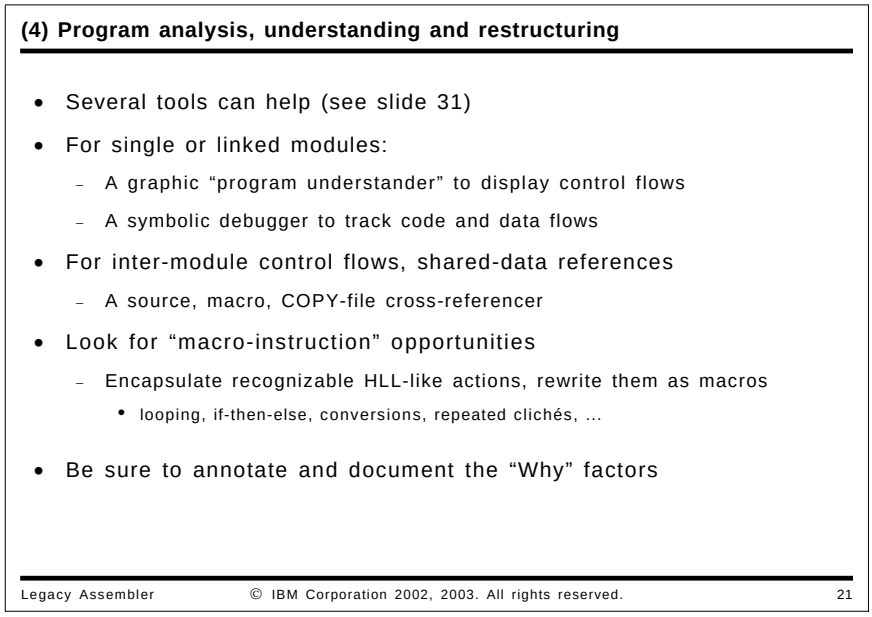
During reorganization, it often helps to ensure that interfaces to system services (such as I/O, date/time, storage management) are isolated to a single module. Specialized skills may be needed to convert such functions, so putting them in a single module will help simplify conversion of the others.

Feldman and Friedman [3] observe that the knowledge required for conversion “includes knowledge of general programming constructs, data types, algorithms, and programming clichés. However, to use the meaning of names one must have understanding of technical terminology, [application] domain knowledge, and some form of natural language understanding, which is difficult in itself.... The same applies to documentation. Moreover, variable names and [existing] documentation are not necessarily correct. Another kind of knowledge that might be useful to a human programmer is an understanding of the program's goal.” Such knowledge may only be implicitly understood, as “company lore”.

The effort involved in restructuring and documenting the application is valuable in its own right, and may provide sufficient reasons to entirely avoid rewriting the application. We will discuss this possibility in “Scenario 5: Modernize and maintain the application” on page 35.

Planning for conversion must therefore include estimates of the pre-conversion effort needed to achieve satisfactory convertibility. “In most conversions, restructuring is prerequisite to what we call the syntax swap, where we swap the syntax of the precooked, restructured original code with the target syntax. This is a relatively easy step. The swapped programs are usually ugly, so another heavy restructuring phase in the target language is necessary to make the new syntax look as much as possible like native code.” [13]

Program analysis, understanding and restructuring



Understanding any program can be difficult. As Brooks [1] notes, “...software is very difficult to visualize. Whether one diagrams control flow, variable-scope nesting, variable cross-references, data flow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant.”

Johnstone [6] recommends understanding control flow as a starting point: “The analysis of assembly code to provide a high level control flow view in terms of the usual high level looping and selection constructs is of great assistance to high level language programmers who are attempting to understand and port low-level code as part of a system re-engineering project.”

Because manual analysis of Assembler Language programs can be very tedious, you should utilize available tools and products to help. Several of these are described briefly in “High Level Assembler Toolkit Feature” on page 40.

- For individual modules, or sets of modules linked together, you can view control flows within and among modules using the Toolkit's Program Understanding Tool. It allows you to view the flow graph at different levels of detail and magnification, and at the same time view the source statements corresponding to each node in the graph.

  While a graphic program understanding tool does not show data flows, you can trace execution-time behavior of both data and instruction flows with an interactive debugger such as the Toolkit's Interactive Debug Facility, which provides fully symbolic displays of source code and variables.

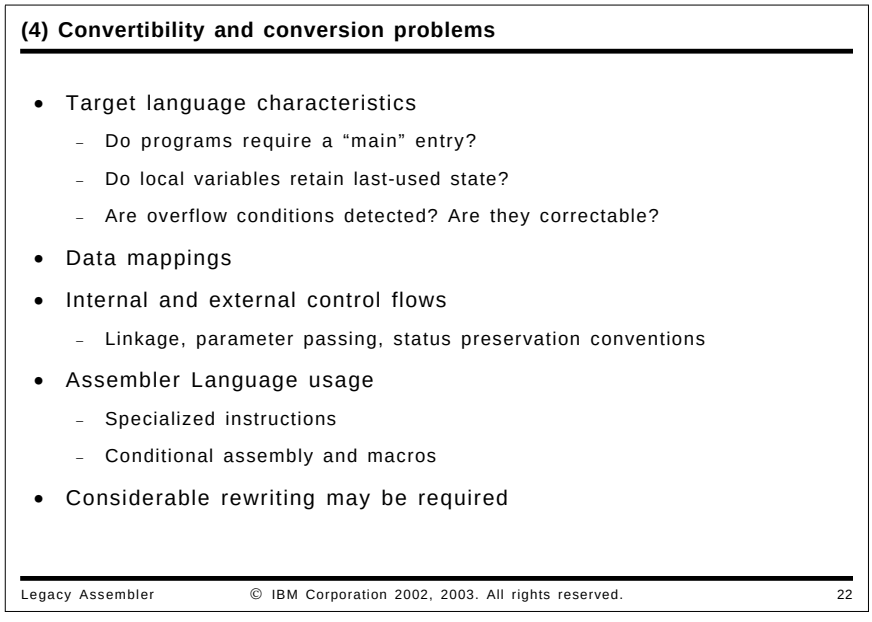
  A typical low-level control and data flow analysis "(as recognized by conversion tools and flow-graph displays) doesn't contain information about the many levels of application, module, and program design" [23] so you will want to add information about the overall intent of the application to its documentation.

- For multi-module application suites, a general source code cross-reference utility (such as the Toolkit's Cross-Reference Facility) can scan source and macro libraries and report inter-program references, uses of symbols; this can be especially important when multiple modules share references to common data structures using the same names.

  Because such "inter-module reference tools provide only minimal high-level 'linkage' connection information, but not information about why" [24] you will want to add documentation describing the purpose and function of each module being analyzed.

A very useful aspect of this application analysis is to identify commonly repeated code sequences, such as loops, if-then-else branches, flag setting and testing, data manipulations, and the like — that is, "programming clichés" — typical of the application. If these are carefully annotated, or even better, rewritten using macro instructions [32,33] they will be much easier to rewrite in the target language.

Convertibility and conversion problems



Many factors must be considered before attempting a manual or automatic conversion. For example, code may need "preparation":

- data mappings may need clarification or reorganization
- control flows may need modification to be recognizable

- dependences on the host system (hardware and operating system) may need to be localized or removed
- subroutine and external-program linkages may have to be modified or standardized
- conditional assembly statements and macros may need change
- peculiarities of the “target” language may need accommodation.

This effort should be included in estimates of conversion costs.

Feldman and Friedman note that “[direct translation required] extensive re-writing of the original assembly-language sources... eradication of techniques that have no parallel in high-level languages... and the use of macros for structured programming constructs.... Many patterns of coding were outlawed, and others had to be replaced by special macros. The result was that the complete sources had to be carefully inspected and extensively modified, requiring a heavy investment of painstaking (and boring) effort by the original programmers.”

- “The existing code often embodies numerous hidden assumptions about its environment; these assumptions need to be discovered and modified in order to allow porting to other environments in which they do not hold.”
- “Some manual preparation of the assembly-language sources was clearly necessary, since they contain self-modifying code and other non-portable techniques.” [3]

These observations indicate that conversion is a lengthy multistage process, and the costs of each stage must be considered.

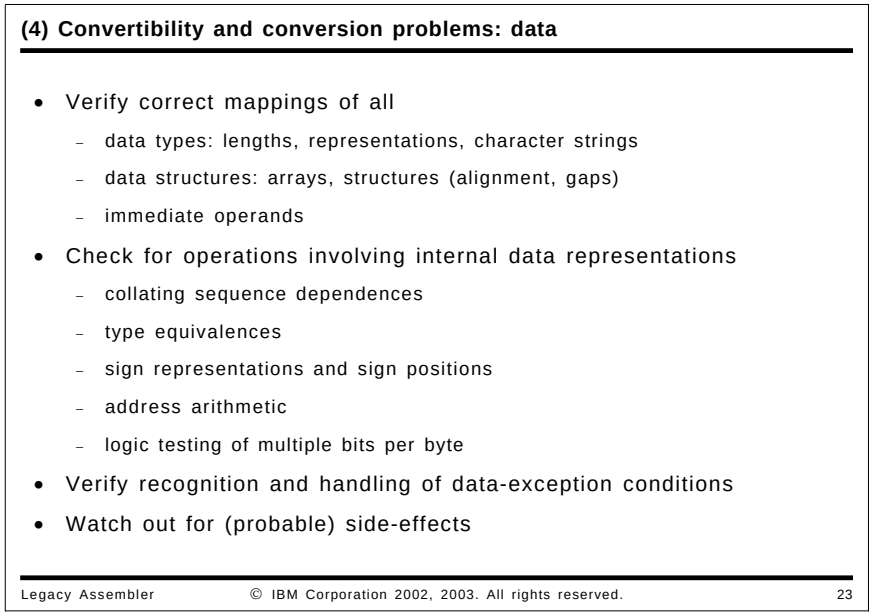
The language to which an Assembler Language program is to be converted may have its own idiosyncrasies that must be accommodated. For example:

- COBOL programs don't necessarily have a main; C programs commonly do.
- The target language may rely on frequent library calls.
- The language may or may not provide controls for overflow checking.
- “Holes” and misalignments in structure allocations may be less evident, leading to data overwriting.
- Pointer code may be machine-dependent, again requiring knowing the characteristics of the target machine's architecture.
- Local variables on a stack may be re-initialized on each entry; but some languages require retaining the last-used state for procedure variables.
- Declaring variables “register” in the C language requires hardware knowledge, and may have a possible impact on the compiler and its optimizations.
- Every language has its eccentricities, some of which may not be evident until long after substantial effort has been spent on a conversion.

Lemieux [8] provides additional discussion of these points.

Evidently, there are many reasons why accurate behavioral equivalence is difficult to achieve.

Data



Correct mapping of data types and layouts is often critical to reliable conversion, and may be difficult to achieve: "...programming languages usually have idiosyncratic data type conventions." [13] For example, data differences include different representations of integers (different default lengths) and characters (C's null-terminated strings, PL/I's length-prefixed strings); immediate operands in IBM mainframe instructions; array mappings; signed vs. unsigned arithmetic; and implicit or expected type equivalences (e.g. between integers and addresses or between integers and characters).

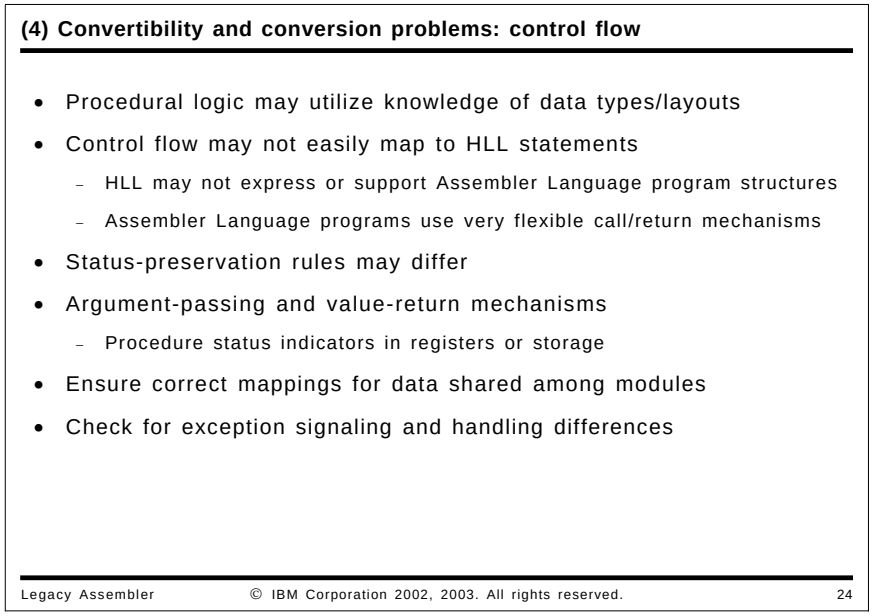
Data mappings may be especially difficult to convert if the target HLL and machine have different representations for internal data types.

"To preserve the program's semantic correctness, the target types ought to have the same internal representation as the source ones." [13] If the internal representations differ, many problems will arise that must be painstakingly detected and corrected. Similarly, you should not expect that similar but not identical data representations will work: "...it is dangerous to map data types of one language to an approximate data type in the target language...." [13]

Type equivalences are another source of problems: "Special difficulties are caused by operations involving the internal representation of variables as well as other operations interacting with memory. ...Solving the problems of unexpected side effects with data types is one of the realities of language conversions." [13]

One of the worst approaches is to emulate an arithmetic construct from the original language in other types in the final language. You must then ask (and methodically verify): "...if we use emulated data types in a native arithmetic construct of a converted program, is the result correct?" [13] Such testing must include boundary and overflow conditions.

Internal and external control flow



Procedural code conversion is usually tightly coupled with the conversion of the data types involved in the procedural code, especially when the procedural code is sensitive to data type representations and layouts.

Conversion tools usually try to identify common control structures (as formalized by advocates of Structured Programming), as well as regular forms of calls among routines. This may not be easy. For example, Johnstone et al. [6] offer this definition: "A reducible structure is one in which all control flow paths enter *via* a single node and exit to a single successor node." They then observe "...we have found assembler programs in which as many as 21% of the functions in an application contain non-reducible control flow. We have also found that the function call structure of these programs is more complex than allowed by most high level languages...."

Johnstone also found that "Control flow in our programs is not neatly partitioned into inter- and intra-procedural components. Compilers typically maintain separate *call graphs* which represent the call hierarchy between functions and *flow graphs* which represent the flow of control within individual functions. ...assembler programs often intermix calls and jumps to the same parts of the application.... A casual examination of hand-written assembler code shows that it is not safe to assume that all functions are single entry." [6]

Other details are important: programs have "meta-behaviors" such as return codes that may or may not be converted correctly.

"Internal subroutines usually employ idiosyncratic conventions about passing arguments and results in registers, and about which registers are saved across calls." [3] A conversion tool may have difficulty recognizing these situations and converting them to equivalent HLL code.

Assembler Language coding provides many choices among efficient program structures. For example, selecting a choice may involve a sequence of IF/ELSE groups, or a "switch" structure; the intended target HLL may not provide a similar choice of structured-programming constructs of equivalent efficiency.

The IBM mainframe architecture "...allows the target of a jump or call to be held in a register: a so-called *indirect jump*. Such instructions are potentially the source of great difficulty when [converting or] reverse compiling because they allow the run-time computation of a control flow

target." [6] Such coding may be the best form of an Assembler Language programming solution, but may have no HLL equivalent.

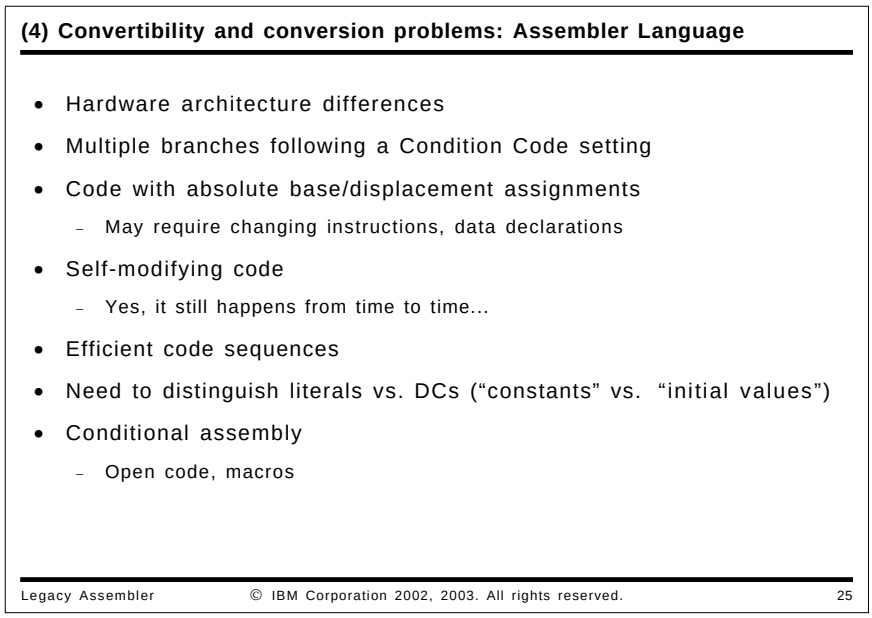
Internal procedures may provide status indicators in registers or flags in storage, or by returning special values; these may create problems if the target HLL has no equivalent capability.

Conversion tools and processes typically operate on one source module at a time, and have difficulties in determining interactions among multiple separately assembled modules — and Assembler Language applications are rarely single modules. Any sizeable application will, for example,

- share external data structures
- call among modules passing arguments by value, reference, or in registers
- rely on special modules for services and error handling.

Such behaviors must be accounted for carefully.

Assembler Language itself



As noted previously, conversions are far more difficult if the original and new hardware architectures differ. For example: moving from a register-based to a stack-based machine; from a flat memory to a segmented memory architecture; and so on. Data representation differences may also complicate matters, so for sake of simplicity we assume conversion is within a single architecture family. (See [3] for further discussion.)

One case in which converted code is likely to be duplicated is when the Condition Code is tested twice (or more), and the predicate being tested does not involve a side effect or a lengthy computation. [3] That is, subsequent branches may test it more than once. For example:

```
- - -          a computation that sets the CC
BP    High     branch if positive
BZ    Zero     branch if zero
BM    Minus    branch if negative
```

Higher-level languages test only one "condition" in a conditional statement; a converter may need to replicate the computation that set the Condition Code for each branch condition. If the computation has side effects, the conversion may be especially difficult.

Some Assembler Language coding techniques can impede conversion:

- If one has coded absolute base-displacement addresses instead of using symbolic names, changes will undoubtedly be required to both the instructions and to the data declarations.
- “Certain assembly-language programming techniques, such as self-modifying code, have no parallel in high-level languages and are even bad practice in assembly language.... Still, there is the question of identifying them....” [3]

Assembler Language programs may have been written to take advantage of specific efficiencies, such as arranging instructions to achieve the best use of the machine's pipeline. Such instruction sequences may be difficult to handle.

A common Assembler Language practice is to calculate a symbolic constant from an expression during assembly time, placing the result in an A-type constant. This may confuse a converter that can't distinguish absolute and relocatable expressions.

Conditional assembly

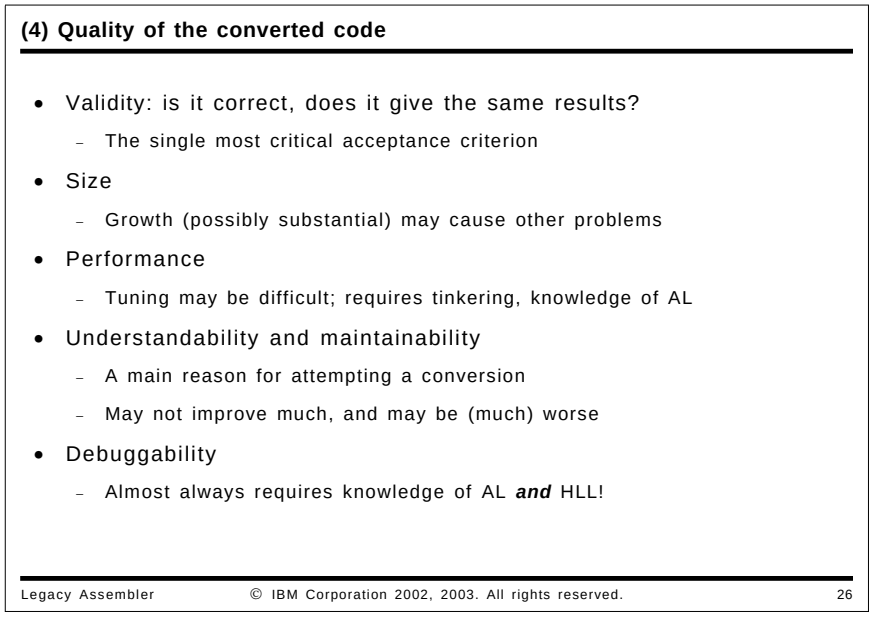
The IBM Assembler Language supports a powerful conditional assembly and macro capability that greatly increases the expressive power of the language. Macros and conditional assembly constructs can be very difficult to recognize and transform to HLL equivalents, even though they may express “high level” concepts.

Feldman and Friedman [3] found “no evidence of handling conditional assembly statements” in their survey of conversion tools, and “solved” the macro problem by working “on the expansion of whatever macros there are.”

Unfortunately, this technique loses the higher-level abstraction embodied in the macro, and may make conversion even more difficult. Going beyond this simple approach (which loses the higher-level intent of the macros) “requires manual analysis of the macro libraries and coding their intent in a suitable formalism.” [3] This extra effort must be considered before conversion.

Macro instructions that support standard structured programming constructs may be difficult to convert to equivalent usage in other languages, even though their intent is usually obvious.

Quality of converted code



Many factors influence the quality of the converted program; we discuss some of them here.

Validity

The most important requirement of code conversion is that the converted programs have the same behavior as the original code. If this is not the case, new problems are introduced. As Voas observes, "'Good' is not a measure of how the code is structured or looks; good is an assertion that the semantics of the function that the code computes is the code's intended function." [14]

Even when programs appear to be converted successfully, checking the behavior of the resulting program may be difficult. "Any language construct that can be abused will be abused. These so-called clever uses of programming language constructs can lead to unexpected discrepancies between the input-output behavior of the original code and the converted code. There are problems associated with overflow, type casting, and other complicated type manipulations." [13]

Similarly, "Optimizing C compilers usually have an 'escape clause' that causes them to ignore aliasing. This can solve the optimization problem, at the expense of potentially introducing subtle compilation errors into the resulting code." [3]

Size

Size *is* important! Code size expansion is normal for compiled code, compared to its Assembler Language equivalent. Even if the performance of the converted code is acceptable, growth in size will impact overall system performance. As Middleton found, "The need for more powerful hardware had been anticipated.... But what had been a surprise was the extent of the knock-on effects. The increased processing capacity required of the computers lengthened the procurement cycle, raised the accommodation costs for the machine and rendered large numbers of microcomputers obsolete." [10]

One successful conversion used an example of a small (textbook) assembler routine that created 90 bytes of object code: when translated to a C program, it occupied 2465 bytes. [3]

Performance

It is reasonable to expect roughly equivalent performance from the converted program, but it is rarely the case that converted code will perform as well as the original. One of the better conversion tools [3] produced "...translated C code [that] was *only three times slower than the original, which in our opinion is quite reasonable for such translation*." (Emphasis added) In addition, as Middleton's study [10] showed: "It was not enough for a language to be available for a particular operating system; it also had to have adequate performance to be usable in practice. [The organizations studied] were therefore required to spend money on substantial equipment upgrades."

Tuning the converted code may be difficult: you will need to examine the generated object code to determine inefficiencies (which, of course, implies a need to understand assembler!). For example, Lemieux [8] provides these interesting suggestions in discussing migrating assembler to C:

- "To maximize the efficiency of programs written in C, think, 'How would this be implemented in assembly language?' Learn how the compiler is implementing the various aspects of C, and use this knowledge when writing C code."
- "Some of the features of C are just not worth the overhead cost in RAM, ROM, and throughput. Learn what these are and avoid them at all costs."

All this is not to imply that there's no escaping assembly-language concerns and knowledge; but equivalent understanding of the performance effects of various HLL constructs may require lengthy experience with the target language's compiler and run-time library, as well as a supply of sophisticated performance-analysis tools.

Understandability and maintainability

One of the most serious problems with converted code is that it tends to look like neither the original nor the target language, but a peculiar hybrid of both. "Although the outcome of conversion should ideally be code that acts as if it is written in the target language and uses the target language's features and idiom, actual converted programs often retain the idiom of the source language...". [13] Thus, you will probably need to know both assembler and the target language to understand the result, which you must expect to be harder to maintain and enhance than the original. Feldman [3] noted "an explanation facility that will allow programmers to understand the relationship between the original source and the translated code was found to be necessary."

A major rework/rewrite effort will probably be required to make it "look like" the new language. Unfortunately, this revision effort may create new bugs and new behaviors that require further work. The better the translator "does its job, the harder it is to understand the relationship between its input and output." [3]

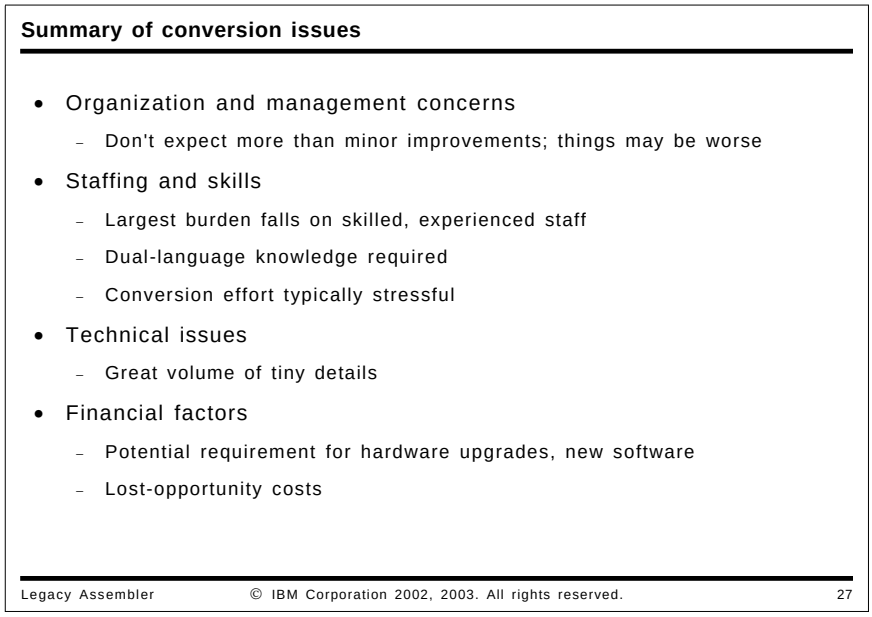
Explanatory comments in the original code may be lost, or have little relevance to the converted HLL code. Cifuentes [2] comments "Human resources are required to enhance the documentation and readability of the recovered code, and lift it up to a maintainable level."

Debuggability

It is a fact of life that programmers often must know the Assembler Language "level" of code to debug it accurately, despite the desirability of debugging at the HLL level. Feldman [3] observes "Another problem we discovered is the difficulty of debugging the code produced by Bogart.... As a result, the original assembly-language programmers found it harder to debug Bogart's code." [3]

Modern debuggers may be able to provide insights at the "high" level of the HLL to which the original program was converted; but since the new program may retain much of the flavor of the original, it may be necessary to know both languages. In addition, HLL debuggers are more complex and sophisticated, which may imply greater development-environment software costs.

Summary of conversion issues



Whether you attempt automated or manual conversion, similar considerations apply. The realities of language and dialect conversion projects can be summarized in Terekhov's five rules of thumb: [13]

- Conversions are difficult.
- Conversions are always more difficult than you think.
- The more semantic equivalence is necessary, the more impossible it gets.
- Going from a rich language to a minimal language is impossible.
- Easy conversion is an oxymoron.

Organizations should not set expectations too high. Researchers and practitioners have reported few benefits from language migration:

- "Given all these problems, we find it hard to believe that converted programs are more maintainable, change-enabled, contemporary, component-based, or any other qualification that is often heard as primary reason for a language conversion." [13]
- Similarly, there may be little difference in maintainability: Wang et al. [15] report that "Maintenance costs are not sensitive to programming language."
- "Few proposals for re-engineering of legacy code have been reduced to practical applications or tools." [3]
- "...we have relied too heavily on the 'language of the day' to solve problems that we could not solve using yesterday's language, and because our problems were never language-related, we naturally failed." [14]
- "The ten organizations that had moved [to a new language] had accumulated experience from this. ...The feelings were generally negative." [10]

Changing to a new language will require additional staff knowledge and effort:

- The task requires precise understanding and documentation of the original code from the highest to lowest levels of detail, and the new code requires exhaustive "equivalence" testing against original code.
- The most experienced and knowledgeable staff will have major responsibilities for the success of the conversion, which may not be the best use of their skills.

- Deep familiarity with both the original and target languages is critical to successful migration; such knowledge is rare, and often under-appreciated.
- "The essence of re-engineering is getting all the gory details right — and the list of annoying issues that must be taken care of is endless." [13]
- "Practice showed that manual work that could not be automatically verified produced some of the most elusive errors. ... This is an important lesson: extensive manual work is not only harmful for the resources it requires, it may also endanger the whole translation enterprise." [3]

The size, scope, and complexity of the conversion task are often underestimated; this may add to the worries of staff, management, and financial controllers.

In addition to language changes, many planning and control mechanisms will need revision. As Middleton found, "Yet the changing of the language used can require alterations to estimating, project management and procurement procedures. This is necessary to work with the new language, but may not bring any new advantages." [10]

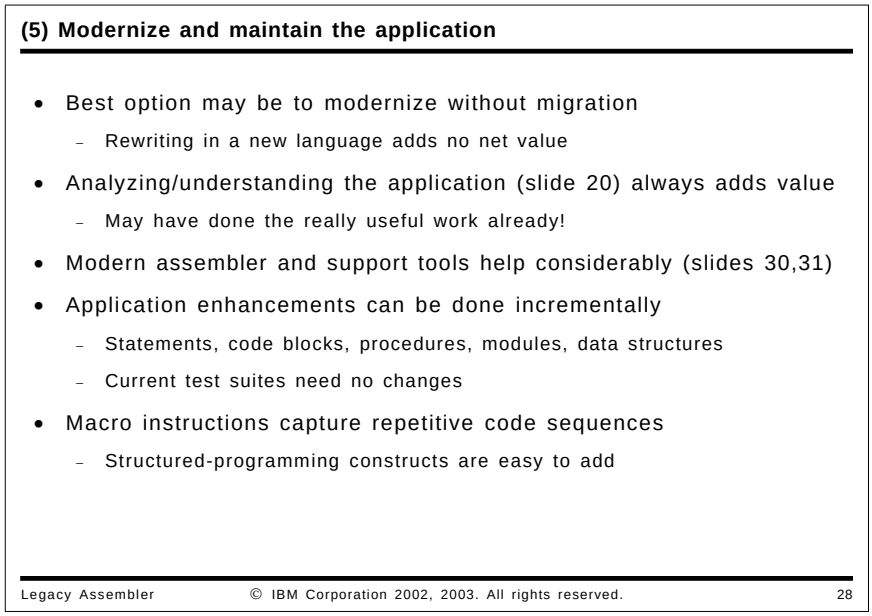
All the factors mentioned previously will have financial impacts, many of which are difficult to estimate (and, possibly, control). One of the least understood is the concomitant loss of enhancement opportunity: original code must be frozen so that regression testing can be done; you can't upgrade the existing code at same time as you are working on revising the converted code. Similarly, conversion introduces new bugs that must be regression-tested, for every increment to every application against the existing version. It is difficult to resist the inevitable temptation to add new function during the changes (which then leads to more testing problems).

If you still want to attempt conversion, the guidelines (on page 16) suggested by Terekhov and Verhoef may be helpful. To which they add:

- "Thus, we hope that more people will accept the realities of language conversions and that decision makers will limit their expectations regarding both the quality and the semantic equivalence of converted code. We also hope that software developers will use our and their own examples as an antidote to the technological quackery of language conversion vendors."
- "As soon as we realize that language conversions are as easy as turning a sausage into a pig, our mind-sets are ready to attack the problem and considerable progress becomes possible." [13]

Scenario 5: Modernize and maintain the application

This scenario is very likely to provide greater benefits and lower overall costs than any of the preceding four scenarios.



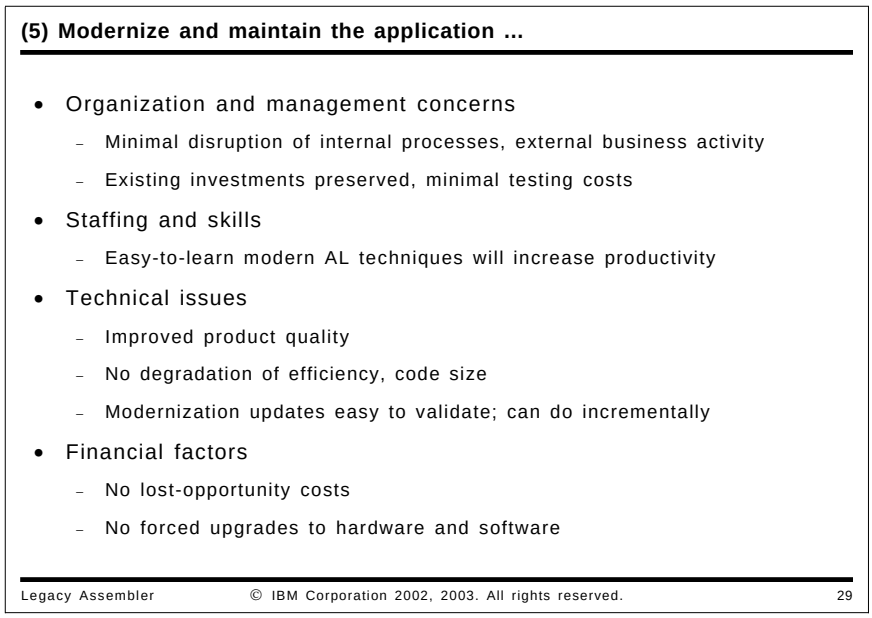
Time and energy invested in understanding and documenting an application are always well spent, even if no conversion is intended. In fact, the restructuring needed for a prospective conversion may be sufficient to make the original code more maintainable! The preparatory actions described in “Preparing for rewriting in a new language” on page 23 may be all that's needed to satisfy most requirements for improved productivity and maintainability.

Modern technologies can help:

- Enhancements to the High Level Assembler and its language can greatly simplify and clarify “legacy” Assembler Language code; we will review these in “High Level Assembler” on page 37.
- New tools supporting Assembler Language can provide insights into application behavior, both for individual modules and for applications involving multiple, linked modules, to assist with many types of reorganization. Some of these tools have wider uses for other languages and file types, as well. “High Level Assembler Toolkit Feature” on page 40 provides a brief overview.

A key factor in modernization is that it can be done *incrementally*: individual statements, blocks of code, procedures, modules, external data, and entire applications can be updated, tested, and accepted in steps as small or large as you are comfortable with; existing test suites need not be changed. Some changes can even be verified simply by comparing the assembled object files, if the generated object code hasn't been changed (except perhaps for embedded time/date stamps).

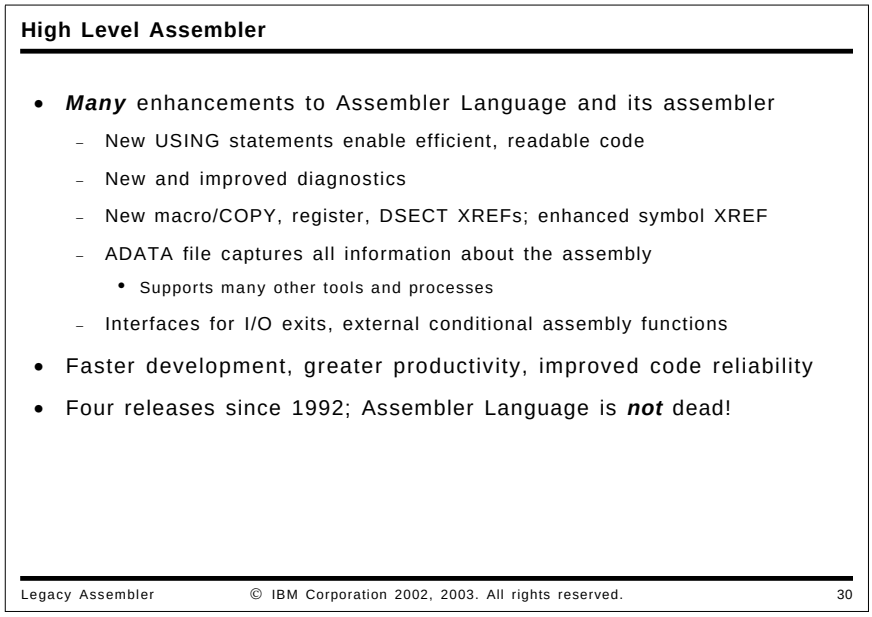
Frequently-repeated code sequences are profitably replaced by macro instructions. These simplify the code while raising the level of expressiveness of those statements.



Modernizing and maintaining an existing Assembler Language application offers many advantages over replacement or conversion:

- Organization and management concerns
  - Existing procedures for planning, estimating, and controlling development procedures need not be changed.
  - Current business activities are not disrupted, and delays in implementing new market requirements are minimized.
  - Existing investments in code, procedures, and skills are preserved.
  - Testing costs for modernized applications are much lower than for converted code.
- Staffing and skills
  - Existing skills are easily upgraded to take advantage of modern Assembler Language programming techniques.
  - There is no degradation in productivity (and morale) caused by having to do “busy work”.
  - Skilled senior programmers can create packages of application-specific macro-instructions that greatly simplify the ongoing work of application developers.
- Technical issues
  - Long-term benefits from improved documentation and application restructuring include better code quality, understandability, flexibility, and maintainability.
  - There are no changes to application size or efficiency due to the behavior of compilers and run-time libraries. A related benefit is that there is no impact on the applications of new operating system and language-product releases.
  - During the period of program modernization and restructuring, changes to the code are very easy to validate, and can be made incrementally.
- Financial factors
  - Because the current application can be kept “in place”, there are no lost business opportunities caused by language-change requirements to freeze the code during the migration period.
  - There are no “knock-on” costs caused by having to upgrade to new hardware or software.

High Level Assembler



IBM recognizes the continuing importance to its customers of Assembler Language applications and their underlying assembler support. High Level Assembler contains many enhancements specifically targeted to satisfying the needs of developers and maintainers of all Assembler Language applications, especially medium- to large-sized applications.

- The new capabilities of High Level Assembler can provide immediate benefits, with no "learning curve". With a small investment, the other new features can be exploited to provide a wide variety of advantages.
- Every increment in efficiency provided for assembler programmers helps to "leverage" their increasingly valuable skills; High Level Assembler provides extensive enhancements in areas that promote greater usability, efficiency, reliability, and productivity.

Portions of the following material are adapted from an IBM "Red Book" [26] and a SHARE presentation [33].

Key Benefits of High Level Assembler

High Level Assembler provides numerous benefits to you and your organization by protecting your investments in applications, people, and procedures.

- **Protect investments in applications**

  By providing improved technology to support existing applications, High Level Assembler can extend the useful lifetimes of those applications while reducing the costs of enhancement and maintenance.

- **Protect investments in people and skills**

  The knowledge and skills of an organization's application programmers are a valuable resource. High Level Assembler helps to maximize the productivity of your application programmers by relieving them of many tedious and unproductive tasks that it can now do itself.

- **Protect investments in procedures**

  It takes many years to develop efficient and reliable project management procedures such as estimation, tracking, and analysis. High Level Assembler can extend the useful lifetime of these procedures by making more efficient use of the human and system resources.

High Level Assembler provides many extensions to past IBM assemblers. These extensions enable substantial savings in time and human and machine resources and support integration of High Level Assembler into tool and development environments.

- Many new and expanded cross reference features are provided. These can significantly shorten the time required to accomplish many of the tasks involved in maintaining and managing applications.

  The features include:

  - A catalog and cross reference of macro and COPY-member usage, source-file record origins, and additional information in the summary statistics allow High Level Assembler to track the precise file or data set origin of *every* input record used in the assembly.

    Note: This feature alone can save hours of expensive and tedious detective work in searching for the causes of versioning problems!

  - A summary of all DSECT definitions.
  - A USING map summarizing all USING- and DROP-statement activity.
  - Many symbol-XREF extensions, including usage tags for modification, branch, indexing, EXecute targets, and for appearance in USING and DROP statements.
  - A register XREF that shows all explicit *and* implicit uses of the General Purpose registers, including usage tags similar to those in the symbol XREF.

    Note: These new XREF features eliminate the tedium in hunting for the few instructions that might have changed the contents of a register or the value of a variable, or for variables that might be part of an addressing expression.

- Many large and small usability enhancements ease application development and maintenance, such as USINGs-in-effect headings on each page that greatly improve the understandability of the code on each page of the program listing.
- High Level Assembler provides many enhanced and new diagnostics. These speed the processes of locating and correcting errors as well as avoid other possible sources of error. They also reduce the technical burdens on individuals with Assembler Language expertise.

  More information is provided for many existing messages. For example, every diagnostic may be accompanied by a second message identifying the file and ordinal record number from which the flagged statement was taken.

- High Level Assembler provides numerous language enhancements that materially improve the speed and accuracy of application development and the quality and reliability of the resulting code. These include:

  - Three new USING statements, which improve program reliability and efficiency and let you manage complex data structures far more simply, naturally, and effectively than was possible with previous assemblers. Surprising as it may seem, these new USING statements allow you to write *more efficient code* than you could without them!
  - Many new system variable symbols provide added powerful function and richer access to properties of the assembly-time environment, and facilitate the tailoring of applications to specific requirements. They also let you easily capture useful information into the object code.

- Existing assembly-time options have been extended and enhanced, and many new options have been added. These options allow increased flexibility and precision in controlling the processes you use to manage application development.

High Level Assembler provides facilities that support the integration of the assembler into tool and development environments. The most important support features are:

- An optional Assembler Data file that contains data about *every* aspect of the assembly: all input records and their sources, all messages, all symbols, complete cross reference information, and much more. This file can be used as a basic source of data to support a great variety of tools such as analyzers and debuggers.
- Optional user exits are provided for inspecting and monitoring the flow of records to and from all user files. With this feature the assembler can be integrated with librarians, config-

uration managers, and other components that can benefit from direct and immediate interaction with the assembler's input and output streams.

- High Level Assembler supports assembly-time external functions. These provide the application developer with essentially unlimited access to any capability of the assembly-time environment and support the programming of complex assembly-time computations and interactions that would be difficult or impossible without them.

The usability, language, listing, and other enhancements embodied in High Level Assembler make development and maintenance of Assembler Language applications easy and efficient.

In summary, High Level Assembler provides greatly enhanced assembler technology to:

- Preserve investments in code, people, and processes
- Improve support for critical assembler-based applications
- Maximize the productivity of personnel with critical skills
- Support integration of the assembler with modern application development tools and environments
- Help avoid costs of unnecessary conversion of existing applications to other languages.

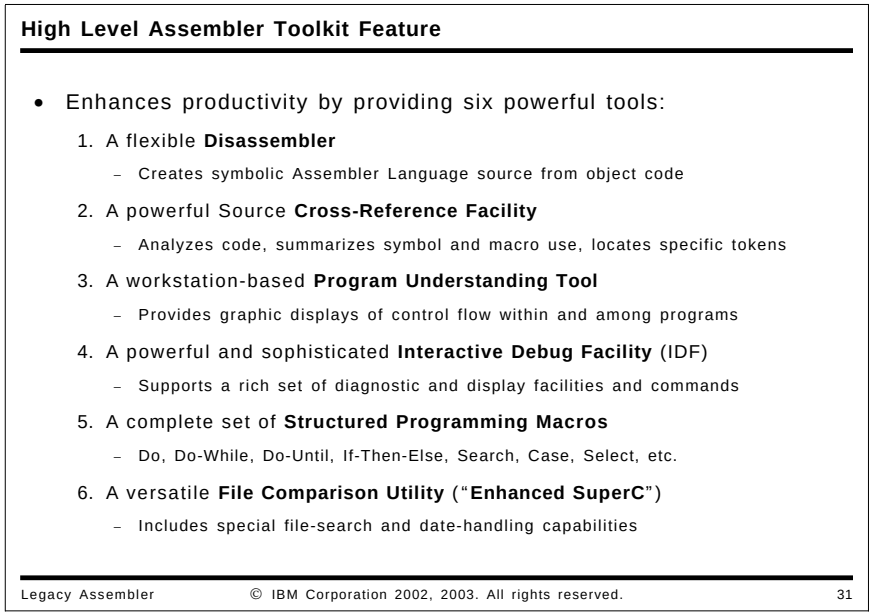
The productivity gains obtained from the use of High Level Assembler can be found in many areas:

- Numerous enhancements to the base language, as well as to the conditional-assembly or macro language improve the readability, maintainability, and efficiency of Assembler Language applications.
- Extensive usability enhancements improve the efficiency and productivity of assembler programmers and maintainers.
- Many new and expanded diagnostic capabilities help in delivering higher-quality and more reliable applications.

Using High Level Assembler will result in cost savings attributable to:

- Faster development cycles due to performance and usability improvements
- More maintainable and reliable applications due to new language, diagnostic, and extensive support features
- More efficient and productive programmers.

High Level Assembler Toolkit Feature

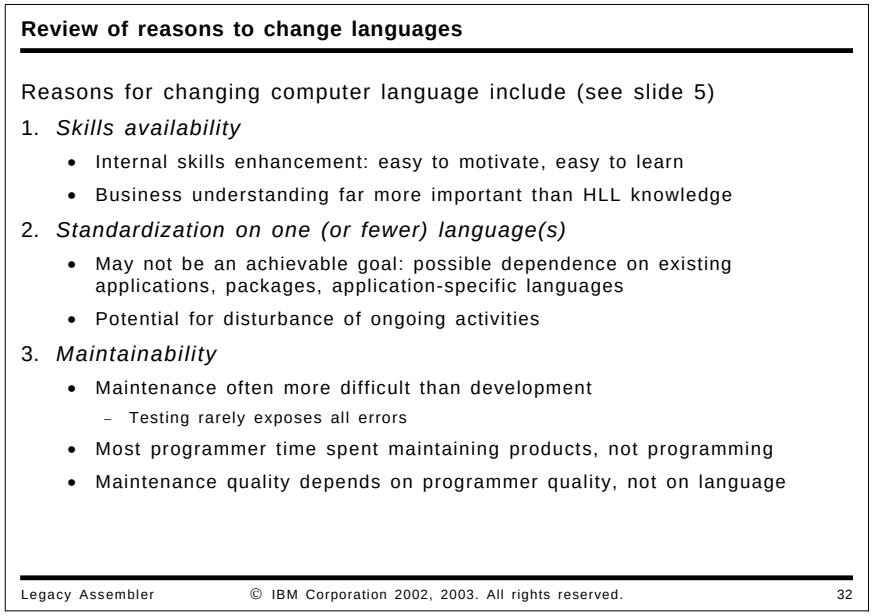


The High Level Assembler Toolkit Feature is an optional, separately priced feature of High Level Assembler for MVS & VM & VSE. It provides a powerful and flexible set of six tools to improve application recovery and development, and to assist in program preparation, analysis, debugging, and maintenance on z/OS*, z/VM*, OS/390*, MVS/ESA*, VM/ESA*, and VSE/ESA* systems. These productivity-enhancing tools are

- **Disassembler**, a tool which converts binary machine language to Assembler Language source statements. It helps you understand programs in executable or object format, and enables recovery of lost source code.
- **Cross-Reference Facility**, a flexible source-code analysis and cross-referencing tool. It helps you determine variable and macro usage, track shared external data structures, analyze high-level control flows, and locate specific uses of arbitrary strings of characters.
- **Program Understanding Tool**, a workstation-based program analysis tool. It provides multiple and "variable-magnification" views of control flows within single programs or across entire application modules. It also links graphical and source views to help you navigate through complex logic.
- **Interactive Debug Facility**, a powerful and sophisticated symbolic debugger for applications written in Assembler Language and other compiled languages; full source-level debugging is supported for Assembler Language programs. It simplifies and speeds the development of correct and reliable applications. (It is not intended for debugging privileged or supervisor-state code.)
- **Structured Programming Macros**, a complete set of macro instructions that implement the most widely used structured-programming constructs (IF, DO, CASE, SEARCH, SELECT). These macros simplify coding and help eliminate errors in writing branch instructions.
- **File Comparison Utility** (known as "Enhanced SuperC"), a versatile file searching and comparison tool. It can scan or compare single file or groups of files with an extensive set of selection and rejection criteria. Typical uses include comparing an original source file with a modified source file, or a pre-migration application output file with a post-migration output file. Additional functions include "smart comparisons" of date fields.

Together, these tools provide a powerful set of capabilities to speed application development, diagnosis, and recovery.

Summary observations



Why Change Languages?

As noted on page 6, an organization might offer these reasons to undertake a programming language conversion:

1. **Skills availability**. Although younger programmers are often not trained in “legacy” languages, many education resources (including internal training) can provide the needed background. This approach may be much simpler than trying to hire people with existing “legacy” skills: motivated people can learn new techniques quickly, and are worth far more than they cost. (The converse implication — that inexpensive programmers may not produce in proportion to their lower cost — is not widely appreciated.)

   Students learn “hot” languages — partly because academics are driven by a need to appear to be on a leading edge — and partly because they may believe that fortunes are to be made working with new and exciting technologies. (A slowing economy will probably prompt programmers to look for security rather than immediate fortunes.) Many currently popular languages are designed to support quick and easy solutions, which may lead young programmers to believe that all programming should be quick and easy.

   Often, the programming staff's main value to an organization is their understanding of the organization's business, not their understanding of the language in which the applications are written: problem-domain expertise takes much longer to learn than a programming language. Whittaker adds, “... the technical work performed in understanding the problem and solution domains can be the difference between project success and failure.” [17]

2. **Standardization** on one language. The goal of one language (or fewer) appears difficult to achieve. The findings of Middleton's research [10] “...illustrate that in this sample of organizations the key motivation for changing computer language was standardization. The use of only one language was expected to improve staff performance when developing and maintaining software. While this is a reasonable assumption it is simplistic. Boehm observed that three things would hamper the... desire to standardize...

   - the need to maintain existing systems written in other languages,

- the use of packages written in other languages and
- the need for specialized languages for certain applications.

This indicates that the goal of standardization may not be a practical one."

Middleton also noted: "Standardizing on a new language also occurred at times when there were other changes such as staff moves and new application areas were being tackled. This added to the uncertainty of an already uncertain environment. This did damage to some projects and must be seen as an additional cost of changing languages." [10]

3. **Maintainability**. "All successful software gets changed. Two processes are at work. First, as a software product is found by people to be useful, people try it in new cases at the edge of or beyond the original domain. The pressures for extended function come chiefly from users who like the basic function and invent new uses for it." [1]

   Prechelt's study concluded that programming languages are not a key factor: "No unambiguous differences in program reliability between the language groups were observed." [11]

   Glass [5] observes "Maintenance typically consumes about 40 to 80 percent (60 percent average) of software costs. Therefore, it is probably the most important life cycle phase." He adds: "Enhancement is responsible for roughly 60 percent of software maintenance costs. Error correction is roughly 17 percent. So, software maintenance is largely about adding new capability to old software, not about fixing it."

   A surprising aspect of maintenance activities is the minor influence of experience: "...one could not, except for corrective, small and simple maintenance tasks, have more confidence in the predictions of an experienced maintainer than the predictions of an inexperienced maintainer." [7]

   The expected improvement in maintainability (by using HLLs) may not happen, as shown by Wang's study [15] comparing maintenance of assembler and C programs.

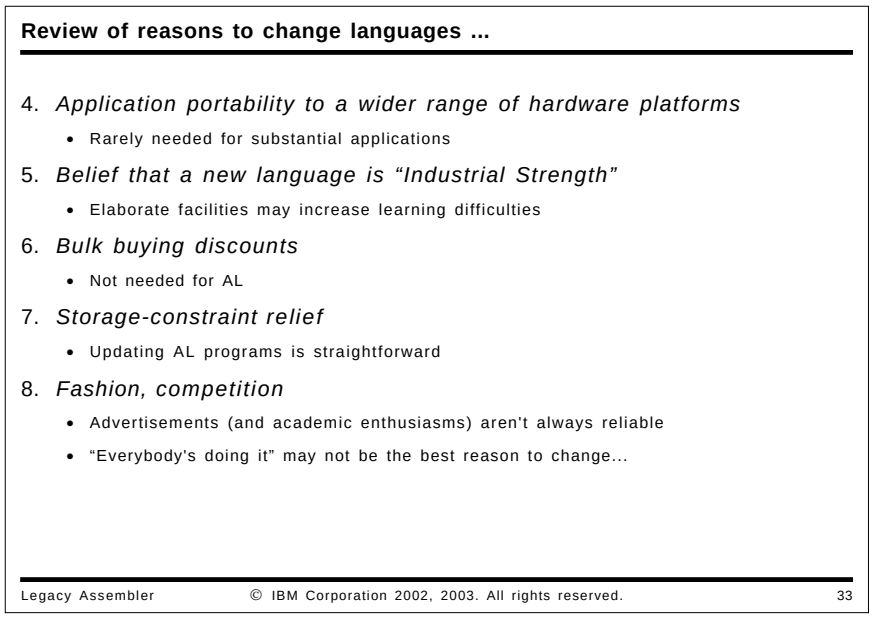
   - There was no significant difference between the maintenance of assembler files and C files. Also, there was no significant difference between the versions written by the original developers and those written by maintenance programmers not involved in the original development. The differences between individual programmers were very highly significant.
   - "Everybody knows" that assembler encourages programmers to use common coupling all the time, whereas C promotes better software engineering practice. On the contrary, we believe that our result implies that good programmers follow good software engineering practice, and conversely; programming language is not important in this context.
   - "Everybody knows" that the "real" reason that maintenance is hard and time-consuming is that maintenance programmers are not involved in developing the original product. They therefore do not understand it completely and, as a result, the maintenance they perform is "inferior" to that of the original programmers. On the contrary, our result is that statistically there is no difference between the two groups.... This could again mean that good programmers follow good software engineering practice, and conversely. Many organizations assign their weaker programmers to maintenance, and that could explain why repeated maintenance can eventually lead to poor quality code.
   - ...the most important factor... has been the skill of the individual programmer. Software engineering education and training appear to be of key importance for the successful repeated maintenance of a software product. Many software organizations leave maintenance to their less experienced and skilled personnel. Based on the results of our research, we believe this is inadvisable.

   The importance of this last point should not be underestimated; similar skills are needed for both development and maintenance. As Glass [5] observes, "Most software development tasks and software maintenance tasks are the same — except for the additional maintenance task of 'understanding the existing product'. This task is the dominant maintenance activity, consuming roughly 30 percent of maintenance time. So, you could claim that maintenance is more difficult than development."

   Maintainability is also affected by the thoroughness of testing, which normally will detect only some subset of the latent errors. Glass [5] provides these cautionary observations: "Software that a typical programmer believes to be thoroughly tested has often had only about 55 to 60

percent of its logic paths executed. Automated support, such as coverage analyzers, can raise that to roughly 85 to 90 percent. Testing at the 100-percent level is nearly impossible; and even if 100-percent coverage were possible, that criterion would be insufficient for testing. Roughly 35 percent of software defects emerge from missing logic paths, another 40 percent are from the execution of a unique combination of logic paths. They will not be caught by 100-percent coverage (100-percent coverage can, therefore, potentially detect only about 25 percent of the errors!). ...There is no single best approach to software error removal. A combination of several approaches, such as inspections and several types of testing and fault tolerance, is necessary."

A programmer commented: "Well-written structured assembler programs are easier to maintain than poorly written COBOL programs. ...[although] it is often claimed that high-level languages have source code that is easier to maintain." [36]



4. **Range of hardware platforms**. While applications written in HLLs are *usually* more easily ported to other platforms, the myriad of tiny details may make this much more difficult than originally thought. "Platform portability is often quoted as a reason to convert, but in reality platforms are seldom changed, and when platform changes do occur, high-level languages are not as portable as you would think, because many platform-dependent extensions are often used." [36]
5. **Industrial strength** of the new language. High-level languages may introduce unneeded complexity (the "Swiss Army Knife" syndrome): "...at some point the elaboration of a high-level language creates a tool-mastery burden that increases, not reduces, the intellectual task of the user who rarely uses the esoteric constructs." [1]

   Complex HLL semantics can actually obscure what's really happening! As Whittaker notes, "A typical C run-time library has over 10,000 functions. C itself has hundreds of built-in functions, operators, and reserved words. Applications routinely use dozens of other libraries of similar complexity. Each of these libraries likely has thousands of bugs." [17]
6. **Bulk buying discounts**. This concern rarely applies to assemblers and their support tools, as they are frequently packaged with the operating system.
7. **Storage-constraint relief**. As mentioned on page 10, enabling Assembler Language programs to use buffers and work areas in 31-bit storage is straightforward.

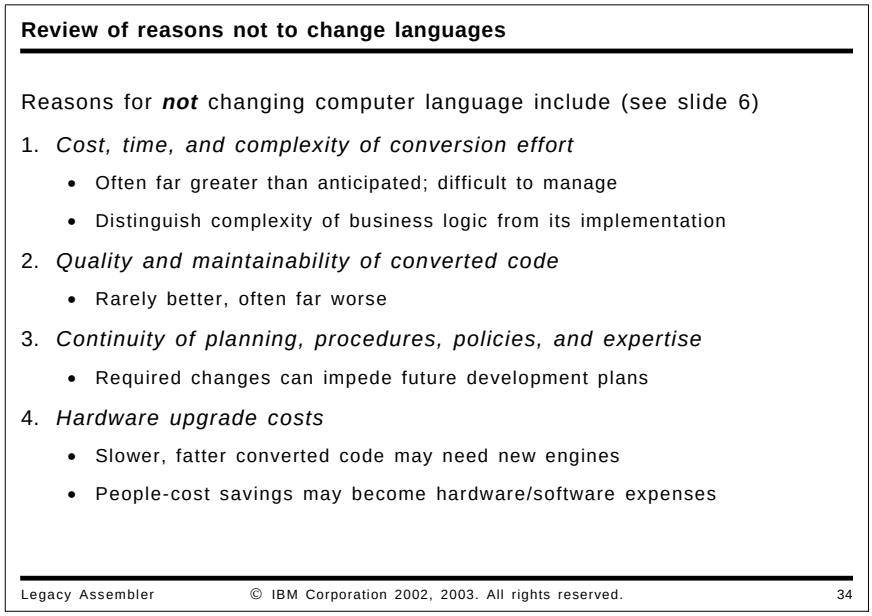
8. **Fashion, Competition**. Software fads have promised much, but delivered little.[1] In addition to their direct expense, some organizations have suffered serious setbacks by investing too heavily in speculative software technologies.

   Holmes [4] comments "...short of any profitability justification, companies invest simply to keep up with rivals." He adds: "Coding schemes... fall prey to the whims and fancies of professional fashion because programming techniques and tools undergo continual development and frequent reinvention." Terehkov adds "The most influential cost drivers of software engineering relate to management, personnel, and team capability — not software tools, although software vendors and academic researchers emphasize them. Many managers become victims of quack software modification tools and practices." [13]

   Whittaker studied software engineering textbooks, and comments: "The problem with current software engineering texts that focus only on design is that they encourage software developers to gravitate to the latest design fads, assuring us that good code will follow. ... We lament the preference for the latest fashionable design technologies and the seeming aversion to the fundamental technical skills such as systems programming and debugging." [17]

   Glass notes: "Most software tool and technique improvements account for about a 5- to 30-percent increase in productivity and quality. But at one time or another, most of these improvements have been claimed by someone to have 'order of magnitude' (factor of 10) benefits. Hype is the plague on the house of software." [5]

[1] Technologies that promised to liberate programmers from their perpetual burdens have included Ada and other high-level languages, Artificial Intelligence, automatic programming, CASE tools (including reverse engineering, business-rules extraction and code generation), causal analysis, chief programmer teams, client-server computing and distributed systems, code inspections, design automation, expert systems, functional languages, graphical programming, object-oriented programming, open source, organizational maturity, packages, process metrics, program proofs and verification, repositories, rule-based programming, Six-Sigma organizations, software metrics, software engineering techniques, Structured Programming, System Application Architecture, test generators, visual programming, workflow process automation, workstations, and Java. While each of these has value and benefits, programming remains obstinately difficult.



Not Changing Languages

As previously noted on page 7, some of the reasons an organization might not undertake a programming language change are:

1. **Complexity of conversion**. Organizations frequently underestimate the difficulty, complexity, and organizational impacts of language changes. [10,13] Whatever the complexities of language conversion may be, it is important to distinguish the *necessary* complexity of the application's business logic and rules from the added complexity of their implementation in a programming language. The form of the latter may obscure the former, but cannot clarify it. As Brooks [1] notes, “The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence. ... The complexity of the design itself is essential, and such attacks [as object-oriented programming] make no change whatever in that.”

   “Not only technical problems, but management problems as well come from the complexity. It makes overview hard, thus impeding conceptual integrity. It makes it hard to find and control all the loose ends. It creates the tremendous learning and understanding burden that makes personnel turnover a disaster.” [1]

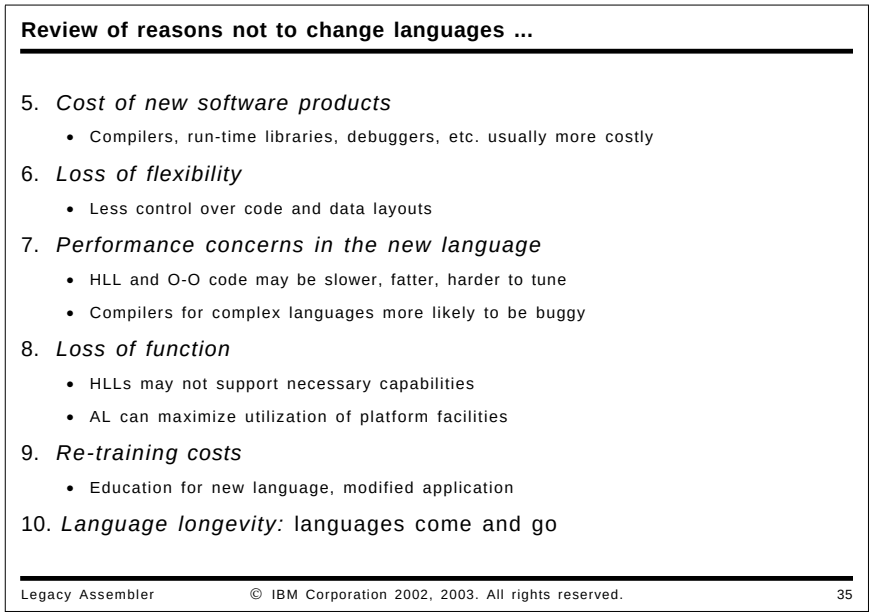
2. **Quality of converted code**. Despite advertising promises of “silver bullet” solutions, code converted from one language to another often resembles neither. [13] The goals of easier maintenance may not be met: the discussion at “Quality of converted code” on page 31 described many relevant concerns. A programmer who understands the original language may not understand the target language; and another programmer may understand the target language but not the original. The two will have difficulty describing and resolving conversion problems.

3. **Continuity** of procedures, processes, and expertise. Middleton's research [10] showed that “...implementing a new language means that the organization's experience of estimating and project management needs revising. This was found to cause severe difficulties for organizations, because by rendering their estimating experience obsolete it meant that mistakes were made when planning for projects using the new language.” Converting development and maintenance procedures to use new design tools, debuggers, test cases, test scaffolds, test tools, code management librarians, and the like will also affect productivity.

Any change is disruptive, and is likely to have adverse effects on development productivity. [8]

4. **Hardware costs**. Higher-level languages tend to be less efficient consumers of machine resources than lower-level languages, increasing both the hardware capital and maintenance costs over the life of the system. “This would increase the total cost of the system to the user, and easily outweigh any savings made during development.” [10] Attaining equivalent performance on cheaper hardware may simply require more hardware.

   A general industry trend has been the displacement of people costs onto hardware costs. It is difficult to assess the net benefits of this trend. As Holmes notes, “...digital technology ‘has directly increased the productivity of direct labor. But these gains are generously offset by an invisible IT overhead.’” [4] This may be due in part to separate budgets for labor and capital-equipment costs.



5. **Software costs**. Higher-level language compilers may cost more than assemblers, and they typically require additional (and higher) charges for their run-time libraries, debuggers, and other support tools.
6. **Loss of flexibility.** Higher-level languages offer less control over code and data layouts, and less control over the generated code, and some functions may not be expressible in the new language. Also, the increased complexity of compilers for “rich” languages means they are more likely to have errors that may cause subtle problems with the generated code.
7. **Performance.** Getting good code from a compiler often requires modifying the source program, as well as care in tuning compiler and run-time options. [8] Ironically, the process of tuning code and options often requires understanding the generated machine language code and its relations to statements in the “high level” language! Similarly, while object-oriented languages can offer simpler programs, their interpreters and run-time environments are usually far more complex than those of procedural languages.

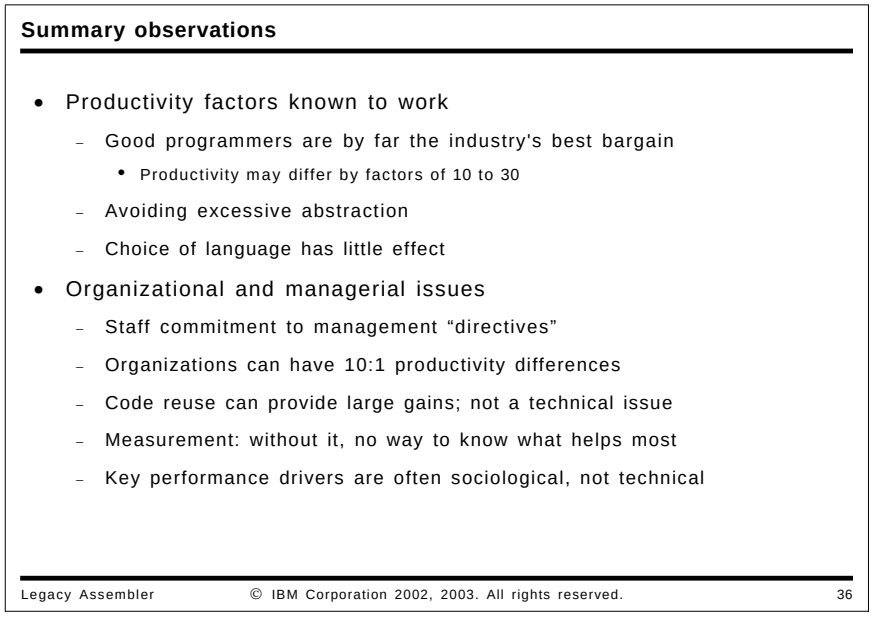
   Programmers may not appreciate the implications of HLL statements; there may be adverse consequences of seemingly harmless choices, “like not knowing the difference between COMP and COMP-3. They had no idea why you would use one over another.” [36] For some applications the impact may be inconsequential, which may lead programmers to believe their coding choices don't matter very much.

   Glass [5] comments “Efficiency is more often a matter of good design than of good coding. So, if a project requires efficiency, efficiency must be considered early in the life cycle. High-order language (HOL) code, with appropriate compiler optimizations, can be made about 90

percent as efficient as the comparable assembler code. But that statement is highly task dependent; some tasks are much harder than others to code efficiently in HOL."

8. **Loss of function.** These concerns include
   - specific instructions: the Assembler Language programmer can often select a single instruction to complete a task that may require complex and inefficient coding, or expensive calls to run-time library services (if they are available at all), in a high-level language.
   - system services: Assembler Language programs can interface directly to a full spectrum of system services that may not be supported by HLLs, or which require substantial run-time library-call overheads to access.
   - excess generality: compilers necessarily handle a wide range of programs and target a very general programming audience, which may make it difficult to optimize or tailor code to specific application requirements. Assembler Language code is easily adaptable to any application need — especially through the use of application-specific macro instructions.
9. **Re-training costs.** Existing staff may require re-training in the new language. These costs must also include the time lost in not working on the new application. "Learning a new tool or technique actually lowers programmer productivity and product quality initially. You achieve the eventual benefit only after overcoming this learning curve." [5]

   Even under the best circumstances, the hoped-for gains from a new language and operating environment may come slowly. As Whittaker notes, "Programming remains monstrously complicated for the vast majority of applications. It is so complicated that developers can work for years in a single language and still not learn all its nuances. Often, developers don't have time to master a language because they must learn a new language every time technology and platforms change. Such change ensures that mastery of language and platform details is not a given...." [17]
10. **Language longevity.** Programming languages are subject to the tides of fashion, and may slowly vanish after a period of popularity. Be careful not to "bet the business" on the latest technology until its durability is proven; programming graveyards are littered with the bones of long-dead languages.



- **Productivity**

  Productivity has many dimensions, of which programming skill is one critical measure.

"Good programmers are up to 30 times better than mediocre programmers, according to 'individual differences' research. Given that their pay is never commensurate, they are the biggest bargains in the software field." [5]

"One rule remains important — as it does for any programming language: without well-trained people you'll never get anywhere; without proper documentation you'll end up not even knowing where you are." (See "Web Sites" item 10.)

"Study after study shows that the very best designers produce structures that are faster, smaller, simpler, cleaner, and produced with less effort." [1] In addition to coding skills, an important ingredient in productivity is a deep understanding of the business: staff having both technical and business knowledge are a critical resource. Whittaker adds: "Problem-domain expertise is intensely technical ... Learning it takes hard work, much study and experimentation, and usually years of experience. Seeding your team with problem-domain experts is one of the best things you can do to increase your chances of success." [17]

The choice of programming language is also not a significant factor: "Interpersonal variability, that is the capability and behavior differences between programmer using the same language, tends to account for more differences between programs than a change of the programming language." [11]

Voas [14] notes that excessive abstraction can be a hindrance rather than a help: "Modern design paradigms that advocate abstraction (for example, information hiding and encapsulation) also make system-level testing more difficult to perform efficiently and adequately. Making systems harder to test will never engender higher-quality systems. Testing is already hard enough!" He adds, "Although there is nothing wrong (and a lot right) with applying formal methods, their limitations have been well publicized and their adoption highly limited. The key problems are that they are hard to implement, expensive, and not foolproof."

Another important factor is testing and error removal: programmers spend roughly three times more time on testing than on programming. "Error detection and removal accounts for roughly 40 percent of development costs. Thus it is the most important phase of the development cycle." [5]

"*Long term productivity*. The key question remains: when the learning required by the change in language had taken place, would productivity rise? The analysis... of how programmers spend their time shows that program writing takes just 13% of a programmer's time. ...This shows that changing the language is unlikely to produce significant productivity gains." [10]

- **Organizational and managerial concerns**

  McKendrick [9] advises "IT managers need to look beyond infrastructure concerns to include process and workflow."

  "*Commitment*. Once a decision was made to standardize on a particular language, there was a sense that developers were to some extent 'let off the hook'. There was not the same responsibility for having to deliver cheap and effective systems. They could not be held accountable for high training costs, technical delays or higher hardware and accommodation costs. A reduction in staff commitment may eliminate gains from staff flexibility increased by standardization." [10]

  Brooks [1] wrote, "The gap between the best software engineering practice and the average practice is very wide — perhaps wider than in any other engineering discipline. ... There is no single development, in either technology or management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity." He goes on to comment: "My first proposal is that each software organization must determine and proclaim that great designers are as important to its success as great managers are, and that they can be expected to be similarly nurtured and rewarded. Not only salary, but the perquisites of recognition — office size, furnishings, personal technical equipment, travel funds, staff support — must be fully equivalent."

  Middleton [10] asked "...what other methods are available to raise productivity? ...the following have clear empirical data to show that they work.

  1. *People*. The estimates converge around a 10:1 difference between programmers. So selecting and retaining good people would raise productivity. (Glass [5] adds, 'The most

important factor in attacking complexity is not the tools and techniques that programmers use but rather the quality of the programmers themselves.')

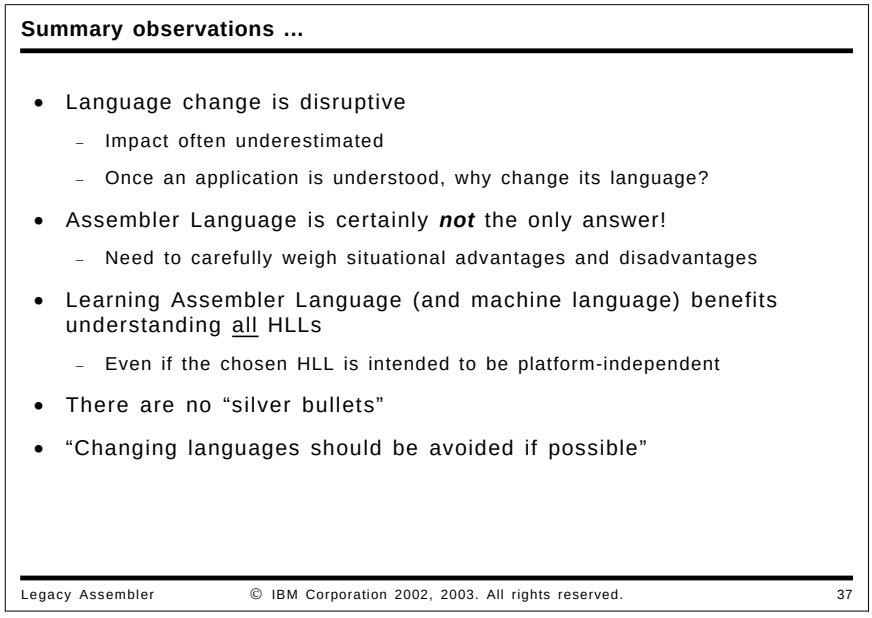
2. *Organization*. The estimates converge around a 10:1 difference between organizations. People of the same capacity are only 10% as productive in some organizations, as they are in others. This difference is due to physical layout, motivation, and organization.
3. *Reuse*. ...companies... report large gains from reuse of software. The problems are not technical but managerial. It is necessary to catalog the software produced and to create a library that is easy to access. (This may not be easy, as Glass notes: 'Reuse-in-the-small (libraries of subroutines) began nearly 50 years ago and is a well-solved problem. Reuse-in-the-large (components) remains largely unsolved, even though everyone agrees it is desirable.') [5]
4. *Measurement*. The simple fact of measuring what is happening provides valuable response and raises productivity. ... no measurements were carried out by the [organizations] that changed their language.

   Research ... indicates that 86% of organizations collect no metrics on their software development process. ... The significance of this is that it indicates that software production is still largely a craft skill, and that productivity is judged subjectively.

All of these four approaches require management skill, rather than crudely attempting to buy productivity with a computer language. The technology-led approach is the equivalent of trying to improve a person's driving performance by buying them a new car." [10]

Middleton concludes, "...the key drivers for organization performance are sociological not technical. Therefore the technocratic response of changing the computer language is not appropriate. Other methods for improving productivity are well documented and proven to produce results, These include policies for selection and retention of staff, organization, reuse of software and measurement of projects. It is concluded that a new language adopted without addressing these other issues is unlikely to be a profitable investment." [10]

A caveat: don't rely on metrics as key measurements. "Because structural metrics do not measure semantics, they cannot say whether the code is good (using this interpretation) — neither can process metrics." [14]



- **Language Change**

  It is optimistic to expect a programming language to provide substantial improvements in productivity, program quality, maintainability, and other measures of "goodness". As Brooks

[1] says, “The hard thing about building software is deciding what one wants to say, not saying it. No facilitation of expression can give more than marginal gains.”

Middleton's conclusions are very illuminating: “The private sector companies [that did not change languages] ... were more sophisticated software developers than the group who had changed their computer language. All this group were initially defensive when asked about their decision to stay with the older language. This may have been because they were afraid of being perceived as being out of touch with new trends.

The results from the ten organizations that changed language recently, points to the following.

- The language used is part of the fabric of the organization.
- Changing it is disruptive.
- The hardware costs that increased more than anticipated were those for purchase, maintenance, and the environment.
- The impacts on project management, development process and estimating were all underestimated.
- These organizations did not collect cost of any data in a coherent form. They were therefore vulnerable to advertisements and salesmen who promised productivity gains.

The results show that increasing staff flexibility was the objective of those organizations that changed language. They were prepared to increase their cost base to achieve this. But their decision was subjective because they had no data on their development performance. The costs of changing language were therefore not fully appreciated, because of their lack of data. They also did not know the critical factors affecting productivity.” [10]

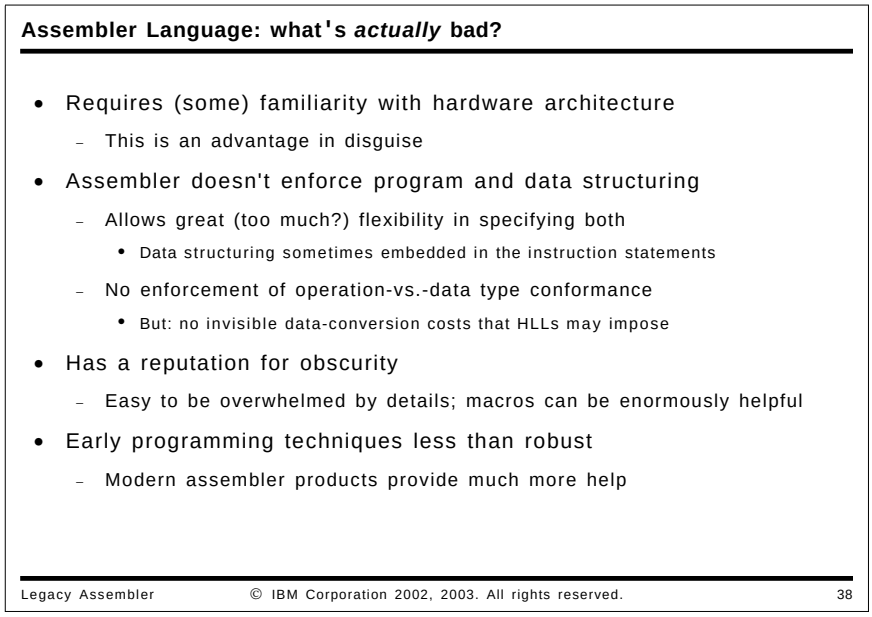
It is worth noting that the effort needed to understand an Assembler Language application well enough to be able to convert it may be applied to modernizing and maintaining it *without* language conversion. As one observer [36] noted, “Sometimes ‘bad old’ is superior to ‘worse new’.”

The final conclusions of Middleton's research paper provide very useful insights: “The research was to evaluate whether changing computer languages had improved the performance of developers. The conclusion was that the key drivers of productivity are social and not technical. The costs of changing languages were higher than expected and changing languages hinders estimating, planning, and project management. Changing languages should be avoided if possible.” [10]

When new applications must be created, there may be no necessary reason to use Assembler Language in preference to a high-level language. One expert (See “Web Sites” item 10) says: “Taking it all together, our standard advice is: do not use assembler when there is no need to. On the other hand, if you have good reasons to use assembler, use it only for those modules that will benefit from it. The largest part of your application is probably best built in your favorite 3GL or 4GL.”

Whatever your choice of implementation language, it is useful to remember that programs generate instructions that execute on a machine, and that knowing how the machine works provides useful insights into the program's behavior. Thus, understanding Assembler Language — even if it won't be the primary implementation language of an application — is an important element of a programmer's set of skills. And, because an understanding of an organization's business is at least as important as programming skills, internal Assembler Language training and education can have enduring benefits; staff turnover can be very costly.

Assembler Language: pros and cons

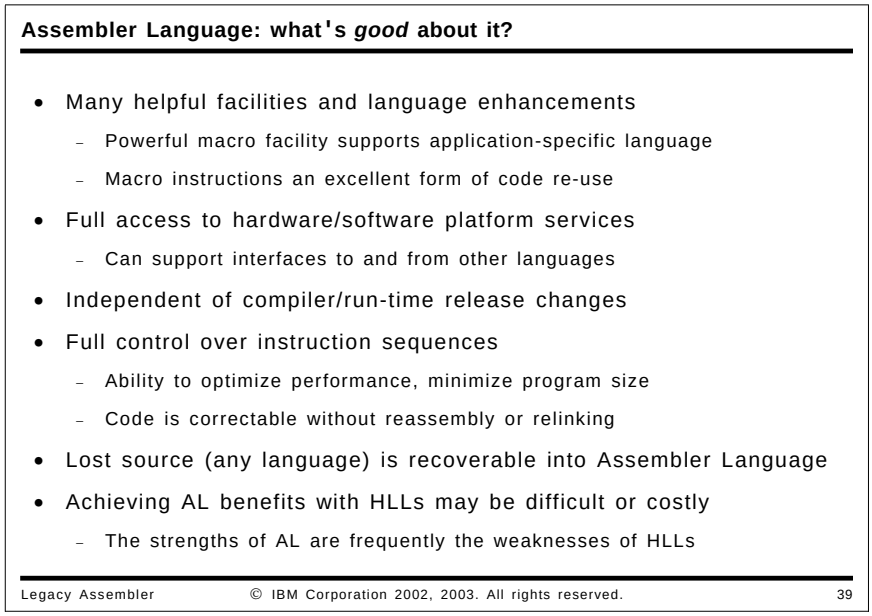


Assembler Language, like any programming language, has shortcomings:

- Assembler Language programming necessarily requires knowledge of the machine's language, in addition to normal concerns with data declarations and program structures. As emphasized many times in this discussion, such knowledge has considerable value even if Assembler Language itself is not being used. (Technical staff cannot do without technical knowledge!)
- The assembler does not enforce any predetermined program or data structuring rules. While this provides great flexibility in specifying both, it also means that disciplined programming styles may be needed to prevent unruly structures.

  Another consequence of this lack of inherent structure is that it's easy to write statements that mix the actions of the instruction with the structuring of the data being operated on. Again, this is easily overcome by careful attention to data definitions.
- The assembler also does not check or enforce any expectations of type conformance between instructions and data. For example, you may (attempt to) perform floating point operations on zoned decimal data. While this flexibility is at times a critically important aspect of the language, there are times when careless errors could be diagnosed if the assembler was less tolerant of such incompatibilities.
- Assembler Language has achieved a (somewhat undeserved) reputation for obscurity. This is due partly to the lack of robust programming techniques in the days when many applications were written in Assembler Language, and partly to the lack of helpful programmer support in the older assembler products.

  In addition to the advantages of training and education mentioned elsewhere, many widely available tools and technologies can help clarify program behavior.



As we have seen, Assembler Language has many beneficial aspects:

- The conditional assembly language supported by High Level Assembler is quite powerful. Using simple macro techniques, Assembler Language developers can build their own application-specific language elements in easily extended increments. Several successful organizations have built major application suites using macro-based application-specific "business languages".[2]

  A well-designed set of application-oriented macro instructions can greatly improve the expressive power of each statement in the language. Effectively, the "high level" qualities of HLLs can be implemented in Assembler Language, and such improvements can be introduced in controllable stages. Such macro instructions can then be used (and re-used) in other applications.

- Modern techniques, tools, and technologies can greatly enhance the productivity of Assembler Language programmers and the quality of their code. The High Level Assembler supports new language enhancements, extensive cross-reference capabilities, and numerous external interfaces that simplify programming tasks.

- Assembler Language provides full access to all the machine instructions and operating system services. This can be especially useful on the System/360/370/390 and z/Architecture CISC architecture, where many powerful new instructions can do the work of previously long and complex instruction sequences.

- Operating system services such as data spaces, multi-tasking, control-block chaining, and dynamic storage and module management can improve usability and performance without major application restructuring, and can supplant previous hand-coded application-specific operations. Note that access to such services is not really an Assembler Language issue; it's often the only language that can provide the access!

- The weaknesses of HLLs are frequently the strengths of Assembler Language: its flexibility makes it easy to write routines that can communicate with other languages and services that may not be natively supported in your favorite HLL.

- Error detection and recovery can be more easily localized to the contexts where problems are most likely to occur, rather than having to rely on a single "global" error-recovery module.

[2] Examples include Allstate Insurance, Sprint, Bell Telephone Labs (SNOBOL), IBM Fortran G, RMFortran.

- Unlike HLL compilers and their run-time libraries, the Assembler Language is stable across releases, meaning that expensive recompilations and regression tests are not required each time a new compiler or library is installed. A new assembler release can be quickly tested by assembling a module and comparing its object file to the current object file: except for things like embedded date/time stamps, the object code will be the same.
- You can write code optimized to your specific needs, without having to "tune" compiler and run-time options to achieve the best the HLL can deliver; compilers cannot know which optimizations, and which parts of the code, are most significant to the application.

  Similarly, "quick fixes" are easily made to Assembler Language programs *without reassembly* by making binary patches ("Zaps") to the executable module. Thus, complex systems can be kept in operation with minimal outages while the source code can be updated and installed in a more leisurely and controlled manner.

  Assembler Language is often easier to debug, especially in cases where the application involves system-level operations or if diagnosis involves a dump, because the object code accurately reflects the source code.
- One little-known advantage of Assembler Language programming is that disassembly tools can reconstruct an Assembler Language source program from object or executable files. This allows recovery and partial reconstruction of lost source programs originally written in any language. (One such tool is described in "High Level Assembler Toolkit Feature" on page 40.)

Recommended Readings

Bibliography: Recommended Reading

1. P. Middleton, **The Costs of Changing to a Fourth Generation Computer Language**, *J. Programming Languages* 2 (1994) pp. 67-76.
2. Andrey A. Terekhov, Chris Verhoef, **The Realities of Language Conversions**, *IEEE Software* Nov./Dec. 2000, pp. 111-124. IEEE Computer Society, Los Alamitos, California.

Other interesting articles:

3. Y.A. Feldman, D.A. Friedman, **Portability by automatic translation: A large-scale case study**, *Artificial Intelligence* **107** (1999) 1-28.
4. Robert L. Glass, **Frequently Forgotten Fundamental Facts about Software Engineering**, *IEEE Software* May/June 2001, pp. 112-111. IEEE Computer Society, Los Alamitos, California.
5. Jeffrey Voas, **Software Quality's Eight Greatest Myths**, *IEEE Software* Sept./Oct. 1999, pp. 118-120. IEEE Computer Society, Los Alamitos, California.

The bibliography on page 57 lists a number of articles, documents, and web site references that may provide useful information. Two in particular are especially relevant to the problems of managing legacy applications: the paper by Peter Middleton [10] and the survey by Terekhov and Verhoef [13] are highly recommended.

Table 1 (Page 1 of 3). Legacy Assembler Language Applications: Summary Observations

Observation	Live with it	Replace with package	Use a HLL converter	Rewrite in a HLL	Modernize & maintain
General Conclusions	Simplest and cheapest short-term solution	Possibly lengthy transition; possible hidden costs	Technically very difficult; probably the worst choice	Harder than it appears; scope and difficulty may be underestimated	New techniques easy to apply; may easily be the best choice
Organizational and Managerial Considerations					
Impact on planning	No change	Must review and fully document existing capabilities to be replaced	Locate and validate quality and utility of conversion tools; staging of migrations	Review staffing, training, coding, testing requirements	No change
Complexity of tasks involved in changing	No change	Depends on quality and flexibility of package	Expected to be very high	Depends on application size, complexity, and documentation	Very low
Time required	No change	Depends on quality and flexibility of package; negotiation effort	May be very long; difficult to estimate	Depends on ability to convert incrementally; potentially long	Relatively little; mainly code “clean-up” and education
Impact on processes and procedures	No change	May be minor or significant; hard to estimate	May be significant; learn a new compiler, run-time, debugger, estimation techniques	May be significant; learn a new compiler, run-time, debugger, estimation techniques	No change
Application source code availability	No change	Ensure access in case of vendor default	Two sets during transition	Two sets during transition	No change
End-user impact	No change	May require extensive retraining	Application stability may be a concern	Intended to be low	No change
Staff commitment to changes	No change	Depends on function and quality of package, level of vendor support	May be low; might blame problems on differences; disruptive change of language “culture”	May be low; might blame problems on differences; disruptive change of language “culture”	Can't decrease; may improve due to enhanced skills and morale
IT Staffing and Skills Considerations					
Skills availability	No change	Depends on requirements for internal support	Need “bilingual” staff during conversion	Need “bilingual” staff during conversion	No change; internal education will help
Skills requirements	No change	Staff involved in acquisition specs; may need to provide internal technical support	Deep knowledge of Assembler Language and target language during transition	Deep knowledge of Assembler Language and target language during transition	No new skills; minor effort to learn new techniques
Staff productivity	No change	Depends on skills transferability to other areas	Likely to be much lower for an extended period	Lower until new technology is learned; little change thereafter	Will improve as new techniques are learned

Table 1 (Page 2 of 3). Legacy Assembler Language Applications: Summary Observations					
Observation	**Live with it**	**Replace with package**	**Use a HLL converter**	**Rewrite in a HLL**	**Modernize & maintain**
Code quality	No change	Depends on vendor responsiveness	Likely to be much lower for an extended period; little change thereafter	Lower until new technology is learned; little change thereafter	Will improve as new techniques are learned
Technical Considerations					
Preparation activities	None	Requirements specs must be very detailed	Code revisions required for convertability; testing to ensure identical execution before conversion	Require high-level and low-level documentation of functions, interfaces, and data structures	Value in having high-level documentation of functions, interfaces, and data structures
Conversion tools and aids	None required; new tools can help with understanding and documentation	N/A	Generally quite unsatisfactory	Mostly manual effort; some assembler idioms may be difficult to express	None required; new tools can significantly improve understanding and documentation
Correctness of resulting code	No significant change; very easy to verify	N/A	Extensive testing required to verify identical results	Extensive testing required to verify identical results	No significant change; very easy to verify
Quality of resulting code	No significant change; depends on how enhancements are made	Vendor responsibility	Generally unsatisfactory; probably very poor	Proportional to time, effort, and skill of rewrite; some assembler idioms may be difficult to express	Can't decrease; may be considerably improved
Efficiency of resulting code	No significant change	Vendor responsibility; typically lower	Likely to be much worse	Probably at least 2-3 times worse (but you may be lucky in some cases)	Certainly no worse; possibly better
Size of resulting code	Depends on scope of required enhancements	Vendor responsibility; typically larger	Probably much larger	Probably larger; depends on skill of rewrite and tuning	Certainly no larger; may be smaller
Flexibility of resulting code	No significant change	Vendor responsibility; expect upgrade charges	Probably very poor	Poor to good, depending on quality of rewrite	Can't decrease; may be considerably improved
Impact of small changes	No change; can often be done with a simple “zap”	Vendor responsibility; expect upgrade charges	Requires recompilation, regression testing	Requires recompilation, regression testing	No change; can often be done with a simple “zap”
Access to system services	No change	Limited by capabilities of the chosen package	Limited by capabilities of target language	Limited by capabilities of target language	No change

Table 1 (Page 3 of 3). Legacy Assembler Language Applications: Summary Observations

Observation	Live with it	Replace with package	Use a HLL converter	Rewrite in a HLL	Modernize & maintain
Debuggability of resulting code	No significant change	Depends on quality of vendor support	Probably very poor	Depends on skills, availability of code documentation and modern debug tools	Certainly no worse; may be much better
Staging of transitions	No change; easily done incrementally	Typically a major cut-over	Potentially difficult; depends on inter-language compatibility of old and new	Potentially difficult; depends on inter-language compatibility of old and new	No change; easily done incrementally
Testing and validation of results	No change	Potentially a major effort	Extremely difficult	Moderately to very difficult, depending on complexity of original code	No change
Impact of IBM product releases	No change; object code always compatible across releases	May require updates as IBM products change	New compiler and run-time require comprehensive regression testing	New compiler and run-time require comprehensive regression testing	No change; object code always compatible across releases
Financial Considerations					
Hardware costs	No change	May increase; vendor software charges may grow with hardware upgrades	May increase significantly; procurement delays likely	May increase significantly; procurement delays likely	No change
Software costs	No change; new tools may be valuable	Expect consulting and tailoring charges	New compiler, run-time, debugger, other tools	New compiler, run-time, debugger, other tools	No change; new tools may be valuable
Staffing costs	No change	IT staff may be reassignable	Need “bilingual” staff during conversion	Need “bilingual” staff during conversion	New staff not required
Staff training costs	No change	Reassigned IT staff will need retraining	Require training in new language, compiler, run-time, tools, processes	Require training in new language, compiler, run-time, tools, processes	Low; can easily teach new techniques “in-house”
Lost-opportunity costs	Maybe none; depends on how promptly enhancements can be made	Probable delays during cut-over	Application code and data must be frozen until conversion is completed and tested	Application code and data must be frozen until conversion is completed and tested	Maybe none; depends on how promptly enhancements can be made

References

Journal Articles

1. Frederick Brooks, **No Silver Bullet: Essence and Accidents of Software Engineering**, *IEEE Computer*, April 1987, pp. 10-19. IEEE Computer Society, Los Alamitos, California.
2. Cristina Cifuentes, Doug Simon, Antoine Fraboulet, **Assembly to High-Level Language Translation**, *Proceedings, International Conference on Software Maintenance*, Nov. 16-20 1998, pp. 228-237. IEEE Computer Society, Los Alamitos, California.
3. Yishai A. Feldman, Doron A. Friedman, **Portability by automatic translation: A large-scale case study**, *Artificial Intelligence* **107** (1999) 1-28.
4. Neville Holmes, **To See Ourselves as Others See Us**, *IEEE Computer* January 2002, pp. 144-143. IEEE Computer Society, Los Alamitos, California.
5. Robert L. Glass, **Frequently Forgotten Fundamental Facts about Software Engineering**, *IEEE Software* May/June 2001, pp. 112-111. IEEE Computer Society, Los Alamitos, California.
6. Adrian Johnstone, Elizabeth Scott, Tim Womack, **What assembly language programmers get up to: control flow challenges in reverse compilation**, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering 2000*, pp. 83-92. IEEE Computer Society, Los Alamitos, California.
7. Magne Jørgensen, Dag I.K. Sjøberg, Geir Kirkebøen, **The Prediction Ability of Experienced Software Maintainers**, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering 2000*, p. 93. IEEE Computer Society, Los Alamitos, California.
8. Joe Lemieux, **410: Moving Efficiently from Assembly Language to C**, *Proceedings of the Embedded Systems Conference Spring 1999*, Maintenance and Reengineering 2000, pp. 1139-1154. Miller Freeman Publishers.
9. Joseph McKendrick, **How Best to Extend Legacy Technologies?**, *Enterprise Systems Journal*, April 05, 2002.
10. Peter J. Middleton, **The Costs of Changing to a Fourth Generation Computer Language**, *Journal of Programming Languages* 2 (1994) pp. 67-76.
11. Lutz Prechelt, **An empirical comparison of C, C/C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program**, *Fakultät für Informatik, Universität Karlsruhe, Technical Report 2000-5*, `http://wwwipd.ira.uka.de/EIR/`, March 10, 2000.
12. Harry M. Sneed, **Using XML to Integrate Existing Software Systems into the Web**, *Proceedings of the 26th Annual International Conference on Computer Software and Applications 2002*, pp. 167-172. IEEE Computer Society, Los Alamitos, CA.
13. Andrey A. Terekhov, Chris Verhoef, **The Realities of Language Conversions**, *IEEE Software* November/December 2000, pp. 111-124. IEEE Computer Society, Los Alamitos, California.
14. Jeffrey Voas, **Software Quality's Eight Greatest Myths**, *IEEE Software* September/October 1999, pp. 118-120. IEEE Computer Society, Los Alamitos, California.
15. Shuanglin Wang, Stephen R. Schach, Gillian Z. Heller, **A case study in repeated maintenance**, *Journal of Software Maintenance and Evolution: Research and Practice* 13 (2001) 127-141.
16. M. P. Ward, **Assembler to C Migration using the FermaT Transformation System**, *Proceedings of the 1999 IEEE International Conference on Software Maintenance (ICSM'99)*, pp. 67-76. IEEE Computer Society, Los Alamitos, California. (See “Web Sites” item 5.)
17. James A. Whittaker, Steven Atkin, **Software Engineering Is Not Enough**, *IEEE Software* July/August 2002, pp. 108-115. IEEE Computer Society, Los Alamitos, California.

IBM Publications

18. **IBM COBOL for OS/390 & VM V2R3 Compiler and Run-Time Migration Guide**, Appendix D. IBM publication number GC26-9764. (See "Web Sites" item 2.)
19. **IBM High Level Assembler for MVS & VM & VSE General Information**. IBM publication number SC26-4943. (See "Web Sites" item 1.)
20. **IBM High Level Assembler for MVS & VM & VSE Language Reference**. IBM publication number SC26-4940. (See "Web Sites" item 1.)
21. **IBM High Level Assembler for MVS & VM & VSE Programmer's Guide**. IBM publication number SC26-4941. (See "Web Sites" item 1.)
22. **IBM High Level Assembler Toolkit Feature User's Guide**, Chapter 2. IBM publication number GC26-8710. (See "Web Sites" item 1.)
23. **IBM High Level Assembler Toolkit Feature User's Guide**, Chapter 4. IBM publication number GC26-8710. (See "Web Sites" item 1.)
24. **IBM High Level Assembler Toolkit Feature User's Guide**, Chapter 5. IBM publication number GC26-8710. (See "Web Sites" item 1.)
25. **IBM High Level Assembler Toolkit Feature Interactive Debug Facility User's Guide**. IBM publication number GC26-8709. (See "Web Sites" item 1.)
26. **IBM High Level Assembler for MVS & VM & VSE Release 2 Presentation Guide**. IBM publication number SG24-3910-01.
27. **IBM PL/I for MVS & VM Compiler and Run-Time Migration Guide**. IBM publication number SC26-3118. (See "Web Sites" item 2.)
28. **IBM z/OS Language Environment Programming Guide**, Chapter 29. IBM publication number SA22-7561. (See "Web Sites" item 2.)

SHARE User Group Presentations (See "Web Sites" item 3.)

29. David B. Cole, **Considerate Programming: Reducing the Maintenance Costs of Commercial Quality Code**, *Proceedings of SHARE 99 in San Francisco, Session 8166*, August 2002.
30. John R. Ehrman, **High Level Assembler for MVS & VM & VSE: Benefiting from its Powerful New Features**, *Proceedings of SHARE 97 in Minneapolis, Session 8165*, July 2001. SHARE, Incorporated.
31. John R. Ehrman, **Finding and Fixing Assembler Language Problems: How High Level Assembler Can Help**, *Proceedings of SHARE 97 in Minneapolis, Session 8173*, July 2001. SHARE, Incorporated.
32. John R. Ehrman, **Assembler Language as a Higher Level Language: Macros and Conditional Assembly Techniques**, *Proceedings of SHARE 95 in Boston, Session 8167*, July 2000. SHARE, Incorporated.
33. John R. Ehrman, **IBM High Level Assembler: Toolkit Feature Technical Overview**, *Proceedings of SHARE 97 in Minneapolis, Session 8166*, July 2001. SHARE, Incorporated.
34. Tom Ellis, Tom Ross, **Assembler Issues when Migrating COBOL and PL/I to LE**, *Proceedings of SHARE 97 in Minneapolis, Session 8216*, July 2001. SHARE, Incorporated.
35. Christine McGarvey, **Assembler 202: Introduction to I/O Concepts**, *Proceedings of SHARE 99 in San Francisco, Session 8155*, August 2002. SHARE, Incorporated.

Discussion Groups

36. ASSEMBLER-LIST@UGA.CC.UGA.EDU, supported by the University of Georgia, at LISTSERV@UGA.CC.UGA.EDU.

Web Sites

1. High Level Assembler: http://www.software.ibm.com/ad/hlasm/
2. IBM Publications: http://www.ibm.com/servers/eserver/zseries/library/
3. SHARE: http://www.share.org
4. Falcon2000: http://www.sapiens.com/falcon2000.htm
5. FermaT (Software Migrations Ltd.): http://www.smltd.com/
6. A2C Software Renovation: http://www.cs.kun.nl/~ hans/A2C/blurb.html
7. Code Discovery: http://www.codediscovery.com/
8. SimoX390 (SimoTime Enterprises, LLC): http://www.simotime.com/simox390.htm
9. Dave Alcock's S/390 Assembler FAQ: http://www.planetmvs.com/hlasm/s390faq.html
10. bixoft: http://www.bixoft.nl/
11. ASM370C (Micro-Processor Services, Inc.): http://www.mpsinc.com/asm370cp.html