RSRE
MEMORANDUM No. 4543

ROYAL SIGNALS & RADAR ESTABLISHMENT

THE EVOLUTION OF Ten15

Authors: D J Tombs & D I Bruce

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 4543

CONDITIONS OF RELEASE

0120150 308892

****

DRIC U

****

DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

Defence Research Agency, Electronics Division
RSRE Memorandum 4543

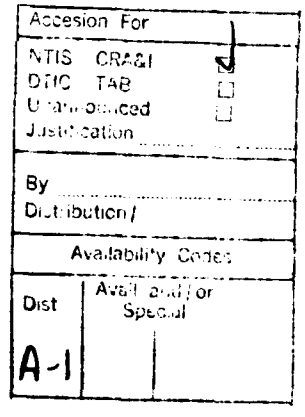
Title The Evolution of Ten15

Authors D J Tombs — Computer Systems Engineering Division
D I Bruce — Signal Processing Division

Date November 11, 1991

Abstract

Recently, development of the RSRE intermediate language Ten15 and its associated demonstration environment was suspended owing to a shortage of resources. This memorandum describes areas for study and potential evolution for future projects having similar goals.



Contents

1 Introduction

Ten15 [1] is a machine-independent intermediate language which has been developed at RSRE to meet a wide variety of programming requirements; in particular it offers a high degree of integrity. Owing to a change of emphasis, the Ten15 programme has presently been discontinued. Instead, effort has been directed towards TDF, a machine-independent representation of program developed from the Ten15 code interface.

This memorandum is intended as a reference for future work programmes which research into some or all of the problems addressed by Ten15. It describes the state reached by the Ten15 project and attempts to highlight those areas still under debate, and to discuss the options for compromise and removal of perceived deficiencies.

The constraints and opportunities for evolution of Ten15 are outlined in order to extend its applicability, both for a wider range of programming styles and for a wider range of machine architectures. The need for evolution is particularly marked with respect to Ten15 support for program analysis and parallel architectures.

The design goals of Ten15 and its relationship with TDF are described in §3. §4 outlines the state of Ten15 at the point of its abandonment, and §5 contains detailed discussion on a number of aspects of Ten15 to highlight perceived deficiencies and possible paths of progress. These latter two sections have been taken more or less verbatim from a paper originally prepared for the IED COOTS project, described in §2, apart from deletion of material concerned only with the COOTS work programme, and minor changes which are alterations to the modal form of the exposition rather than the content.

2 The COOTS project

Ten15 evolution has been under consideration at RSRE for a number of tasks, but in particular as part of the DTI IED3/1/1059 COOTS collaboration, in which RSRE, University College London, and Harlequin Ltd are investigating design and support for object-oriented programming languages and environments for MIMD parallel machines. As part of the RSRE effort Ten15 would have been compared with other implementation techniques; TDF has now been substituted in this rôle. The Ten15 work performed under COOTS is described in [2].

In the first phase of the project we had to identify the extent of evolution of Ten15 needed to provide parallel and object-oriented behaviour. In September 1990, therefore, we prepared a working paper exploring its prospective developments. This described the state of development of Ten15 as it existed at that time, and how it had to be enhanced, both for COOTS and more generally. The majority of the content of the present document is taken from that working paper.

3 Ten15 and TDF

Ten15 is an algebraically defined abstract machine, designed to provide a formal, integrity-preserving, yet flexible and efficient framework, within which a wide variety of software engineering problems can be addressed. One of its major uses is as a common intermediate language between different programming languages and computing hardwares, with high level support for program composition and efficient interworking. The efficiency problems that have bedevilled previous attempts at universal intermediate languages are overcome here by retention of typing and high level constructs. Ten15 is strongly typed, with a uniform

type system that embraces a wide variety of high-level languages, together with types more traditionally appropriate to operating system functions. Dynamic type checking is also supported, but it is the static type checking for strongly typed languages that provides efficiency. The ability to choose between dynamic and static typing gives the greatest scope for combining flexibility with efficiency.

Ten15 code is converted to machine code through a trusted translator that respects the type rules and other semantic properties of the Ten15 formalism. To maintain integrity over the whole system a Ten15 machine comes with a user environment which provides support for programming and system services. These system services are themselves written in Ten15, via the extended type mechanism. Within the environment one can construct and execute pieces of Ten15 but nothing else. Thus, provided the translator is correct the system cannot be perverted and its integrity is preserved.

A few years ago, a machine-independent code interface was introduced to simplify implementation of Ten15. This has since been developed into TDF [3], which has found application as an intermediate representation for a variety of programming languages. At a lower level than Ten15, it addresses questions of language neutrality and code generation more directly. Unlike Ten15 it does not exhibit strong typing or data integrity, nor does it contain explicit OS structures or attempt to be a complete system in its own right.

The twin aims of flexibility and efficiency, within the constraints of formality and consistency enforcement, particularly at the system level, have naturally led to certain compromises and pragmatic decision taking in the design of Ten15. These compromises and decisions need to be questioned both when considering how to develop the formality, and when considering its breadth of applicability. The presence of TDF presents an opportunity to refocus Ten15 more directly on these integrity and system aspects, with code generation and language neutrality being downgraded to a certain extent.

Earlier this year further work on Ten15 in its present form was abandoned, and much software was left in an incomplete state. However many of its design goals are still valid and may be addressed by similar projects in future. It is therefore important to preserve the intellectual effort already put into the project, in such areas as the means of construction, the principles underlying the type system, integrity considerations, and the mainstore memory model.

4 State of Ten15 development

4.1 Ten15 versions

The present definition of Ten15, used in the VAX system and by existing compilers and translators, is known as Version 0 [4], [5]. An extension has been defined, called V0-p, which has added features for pseudoparallelism. V0-p was the original baseline for the COOTS project†.

† The Version 0 baseline:
Definition of sequential V0 — complete
VAX translator for V0 — complete
Experimental kernel and system for V0 running on VAX — almost complete
Definition of pseudoparallel extensions for V0-p — complete
Extension of VAX kernel forV0-p — incomplete

A paper definition, Version 1, irons out oddities in the Ten15 operations and has some extra functional power. In particular, in V1 there is a more fluent model of integer arithmetic, mainstore memory can be addressed at finer granularity, and complex operations such as case-union have a cleaner semantic basis. One improvement is that many operations have an additional form that, on detecting an error, branches rather than raising an exception. V1 is far from perfect, but it is not intended that any further work will be done on it. V1 has a pseudoparallel extension V1-p, which corresponds directly to V0-p.

Ten15 Version 2 is a postulated definition which will have a much more rigorous semantic basis, aimed at achieving greater extensibility and ease of transformation. It may be that several iterations will be needed before we reach our ultimate goal. Much of the research needed for its definition is continuing.

4.2 The demonstration system

The demonstration system is essentially an updated and improved version of the environment written for the RSRE FLEX capability architecture [6], transferred into a Ten15 framework. The first prototype system has been implemented on VAX using VMS and DECwindows, and was in the final stages of development at the point of abandonment. It is anticipated that this system should port easily to other uniprocessor implementations, given appropriate device drivers.

The user-interface to the system is an extensible integrated editor. Rather than having commands and files in the conventional sense, a different, very flexible, style is presented. The emphasis is on construction and combination of values, where any operations that are valid on a value may be performed, giving unanticipated use of program and data.

Within this environment there is great potential for program development. Documents are just another kind of value, and may have arbitrary values embedded within them in addition to text, which can itself be structured in a variety of ways. A compiler is just a program that takes such a document as its input, either producing the compiled result or highlighting any mistakes. A module is an updateable value that contains a compiled unit. The module system tracks dependencies between modules and can automatically provide minimal recompilation to ensure that they are kept consistent when components are altered. Compilers exist for a variety of languages, including Algol68, Pascal, Standard ML and a notation specially designed for Ten15. If a program fails, then the resulting exception value can be diagnosed, enabling the values known to the program at that time to be examined and, of course, operated on in any way the user sees fit.

The current situation is that we can transfer fragments of Ten15 to the VAX and translate and run them there. We do not have a fully usable Ten15-based operating environment available for general use, though many of the components are individually working, such as the evaluation and examination facilities, and the persistent filestore. The major unfinished components are the notation compiler and the module system; major problems have been encountered in bootstrapping this software. Whilst the environment is of significant value, it cannot be considered a true demonstration of the power of Ten15 without full support for these features, or while it is based upon Version 0.

4.3 Enhancement and upgrading

Unforeseen future developments will change Ten15 systems in their breadth of applicability and semantic basis. Ten15 must therefore be capable of evolution and extension. When

extending Ten15, for example to accommodate a new kind of computer architecture or programming paradigm, care will always have to be taken that uniformity is preserved as far as possible. There is the risk of complicating the semantics or compromising the integrity of Ten15 which previously existed. For example, V0-p tasks introduce the requirement of atomicity in updating shared memory. The intent of the Version 2 basis, which places greater emphasis on formal semantics, is to encourage unification rather than complication.

5 *Aspects of Ten15*

This section presents an overview of various aspects of the Ten15 world, with particular reference to their perceived deficiencies. Some deficiencies are of minor impact; for others any solution would demand undue effort or seriously compromise some other aspect of Ten15.

Associated with each aspect is a list of work items to correct the deficiency. No attempt is made in this paper to categorise them in importance or complexity†. Many items are incidental, but some, especially those intrinsic to Version 2 research and the memory model, are central to future development and involve studying a great deal of theory, and so must be considered a high risk. Very little of the proposed work has ever been done. Indeed the expectation that even the minimal developments discussed would be beyond the resources available was one reason for discontinuing Ten15 as it stood.

5.1 *Method of definition*

Ten15 has been defined as an algebraic signature. It is the target, and sometimes also the source, for programs which produce other programs, such as compilers, simulators and optimisers. Apart from Pascal and Algol68 compilers we have hardly demonstrated its power in this respect; SID [7] is one tool which could be retargetted at Ten15. To operate on a piece of Ten15, to use it as a source for a transformation, one constructs a homomorphism over the signature; that is, a set of small operations, one for each constructor of the signature, from which the complete transform gets built automatically. Examples include pretty printers, encodings formatted for disc, and the translator itself. At the moment few such tools have been built, so we do not yet know their potential — perhaps the best model is the VAX translator. The schema for producing homomorphisms has to be delineated better. David Bruce has developed a proper model for such transformations, initially for a translator for the transputer, which, given a set of functions to decompose one concrete representation and a set of encoding routines to construct another, will generate a homomorphism automatically.

Much work remains to be done on the mathematics underlying the Ten15 algebraic definition. It is largely independent of the outside world, apart from the introduction of formal tokens and primitive notions like integers and strings. However, the semantics are described almost entirely in English. To use Ten15 as a formal basis for programming they must be expressed formally. We expect the semantics of V2 will be so defined. Finding a basis for V2 has turned out to be much harder than expected. For example, the semantics and result type of the field selection operator are dependent upon the value of an integer parameter. Present lines of research have recently taken us into lambda-calculus, functional programming and Martin-Löf's intuitionistic type theory.

Should this research yield fruit there would be a large qualitative increase in the means of construction of Ten15. Essentially Ten15 would be self-representable, in the sense that we

† As presented in the original working paper, the work items also contained an analysis of their relevance to the COOTS project.

could build a type system sufficiently powerful to allow construction only of correct Ten15. It would be easy to extend a Ten15 defined thus, and to produce different variants tailored to different machine characteristics. Features which are presently incorporated in an ad-hoc manner, like polymorphism, concurrency and ADTs, could be defined naturally. As a further benefit, the improvement of the semantic basis would facilitate the automated production and verification of tools such as interpreters and translators.

Work items:
demonstrate algebraic construction of Ten15 by programs
demonstrate transformation of Ten15 to a homomorphic image
Version 2 research

5.2 *The translator*

The translator is the program that produces directly executable machine code corresponding to its Ten15 input. It is distinguished in that it is the only program that has the ability to produce such code. The trust in a Ten15 system resides solely in the translator, or equivalently, it is the only program that is permitted to break the integrity of the Ten15 machine, as expressed by the type system. Whilst we do not believe the translator can be formally proven as yet, we consider that the homomorphic form of construction will aid greatly in avoiding errors.

In the context of V2 it might be better to consider whether a symbolic interpreter is a more important program. An interpreter would yield the true operational semantics of Ten15 and overcome the mental distinction between compilation time and run time. There are also several methods of automatically generating a full translator directly from such an interpreter.

Work items:
demonstrate trustworthiness of translator
write an interpreter

5.3 *The type system*

Underpinning all program and data within a Ten15 system there is a strong type system. Each fragment of Ten15 program delivers a value; the only applicable operations are those appropriate to its type, which is a homomorphic image derivable in finite time. The available operations thus define the functionality of the system; by choosing them carefully we have achieved full data integrity, in the sense that no value can be treated as having a type incompatible with the one it had when it was first created. The translator guarantees this by rejecting any Ten15 that does not statically conform to this type regime. With V2 the type system will become even more powerful, and will demand especial attention to its implementation.

It is important for several reasons that the type system and primitive operations should be independent of specific machine characteristics. First, the correctness of a Ten15 program is determined by type, and it would be quite unsatisfactory if programs were deemed correct on one machine, but not on another. For example, in V0 the bounds of integer types are limited by implementation limits (usually the machine word length), and this is reflected in the types of some operations. A program that is type-correct for one machine may therefore be type-incorrect for one having different word length; this is clearly an unacceptable state of affairs.

Second, it should be feasible to write programs that are truly portable, in the sense that their run-time behaviour will be the same on any implementation. In both V0 and V1, for example, equality is defined as equality of representation, so that it is extremely difficult, if not impossible, to reason formally about the behaviour of the program. Clearly, some physical resource limits, such as the amount of available store, cannot be completely avoided, but it might be questioned what, if any, machine-given limits are acceptable. When, for example, the limits of bounded integers are constrained by the implementation, numeric overflow might occur on one machine but not on one having larger word length, even though their types are the same. Obviously, the presence of non-determinism in a parallel system means that we cannot specify a single behaviour. Instead, it is essential that we can reason sensibly about non-determinism, for example, to demonstrate that its presence is benign.

Third, the type system places implicit constraints on an implementation, and we should not casually prescribe any particular representation of its values that might prove grossly unsuitable for some architectures, thus condemning them to poor efficiency. One requirement that is technically essential is that if one type can be coerced to another, then corresponding values of the two types must both have the same representation. Another, this time somewhat spurious, requirement is that integers are virtually required to be two's complement.

The types of the system have been chosen at quite a fine granularity to allow efficient physical representation and coding of operations, but not at the cost of inordinate compile-time complexity. For example, integer subranges have been incorporated, but not subsets. The system also extends beyond the types normally found in programming languages to describe features that are traditionally associated with operating systems. Other type primitives, the array for example, are quite high-level, and could be expressed as an abstract data type. However, the operations on an ADT might not always be as efficient as those on a built in primitive.

There is also the pragmatic concern that run-time overheads due to the advanced features of Ten15, such as first-class procedures and garbage collection, should not be too high for programs that do not use them — including all those derived from languages such as Ada, Pascal, etc. This has influenced the level of data-abstraction so that, for example, standard integer arithmetic is defined to be bounded, with an overflow exception if the machine word length is exceeded, rather than making all integers unlimited.

Ten15 must also be able to support several forms of polymorphism, both on their own merits and to meet the requirements of modern functional languages and object systems. No direct support for ad-hoc variants such as overloading are provided, as they are syntactic niceties that compilers should deal with before generating Ten15. The present type system includes universal quantification, so that algorithms can be abstracted away from unnecessarily concrete instances; existential quantification, which can be used to built ADTs and objects; and some measure of subtyping (i.e., bounded polymorphism), providing a way of tackling some forms of inheritance. These features have been added into Ten15 over time, in a somewhat haphazard manner. We would like to treat this area more uniformly in future, and are not yet certain of the theoretical bounds to higher-order typing. It is an important part of the research for Version 2 Ten15.

A particular problem is that no values in the system can be treated in a completely polymorphic way, in that they cannot be regarded as having completely unknown type. Polymorphism is restricted to the contents of constructs of known size, such as pointers. This is because the size of each value must be known at compilation time, in order that sufficient space can be allocated to hold it. Recently the concept of 'normalization' to a fixed size has been introduced, with the intention that this will eventually be the only fixed size value.

Unfortunately, such normalization tends to pervade the whole program, and there are severe interoperability problems between the non-normalized and the normalized worlds. The alternative, of implicitly normalizing every value (as in ML, Lisp and functional languages), is considered by some as contrary to the spirit of fine granularity; it may also be slightly less efficient.

Work items:
- representation-independent semantics for equality, integer types, etc.
- consider granularity and breadth of scope of the type system
- implement efficient algorithms to manipulate V2 types
- decide whether every value should be implicitly normalized

5.4 *Cyclic structures*

For integrity reasons, every declaration and memory allocation must be properly initialised with a value of the appropriate type. This can, however, cause difficulties with cyclic structures, where the item being constructed is needed in order to construct itself! There are some tightly cyclic types for which there is no satisfactory means of creating any suitable values at all. We need to develop a general model that will adequately address this problem, since these types are otherwise perfectly reasonable, and occasionally prove rather useful.

In V0 there is an operator, parameterised by a type, which creates a valid value of that type, thus solving the immediate problem. Unfortunately, we know that this is not a good solution — especially since, in the more powerful type systems we are interested in for V2, the existence of such values is undecidable. There is also a special mechanism for the most important class of cyclic structures, namely mutually recursive procedures, but again this is not a long term solution. Many functional languages allow totally general recursive equations, but usually achieve this via lazy evaluation, a technique which is, alas, not without its limitations. Other languages have evaded the problem by placing strong syntactic restrictions on the construction of recursive structures.

Work item:
- find a method of constructing general cyclic data

5.5 *Procedure closures*

One kind of data value in Ten15 is the procedure closure, where a block of code is bound to non-local data. This is very important since we need it to build systems. The ion is a generalisation of procedure that has multiple binding times. Procedures and ions are first-class values just as scalars and vectors are; they can likewise be building blocks in composing other values. The procedure is the unit of program in Ten15 — it cannot be broken down to reveal the code and non-locals from which it was constructed, but only applied to arguments to deliver a result. Thus, access to non-locals can be strictly controlled, and is the prime means of achieving data security (as opposed to integrity) in Ten15; for example, secure communication can be implemented using a private non-local as a channel.

We can identify several subclasses of procedure which have particularly simple or useful analytic properties, such as separability (no non-locals accessible from another procedure), monomorphism, referential transparency. No such class has been formally identified in Ten15. Other properties of a procedure concern whether it is suitable for special methods of compilation; for example, removing tail recursion, or treating as a simple subroutine. These

are essentially compiler optimisations (concerned with such details as the form of the procedure body or the lifetimes of local and non-local values) rather than an analytic feature.

Work items:
need for multiple binding times
consider level of data security
analysis of non-locals for shared variables
static analysis to find other properties

5.6 *Interactive working*

We envisage working in a uniform world where the distinction between systems and applications programs is blurred and any value can be created and used freely and in an unanticipated manner throughout the environment. We believe that Ten15 could be used to build such an advanced human-computer interface. Key features of an implementation are a flexible command interpreter capable of producing any desired Ten15 fragment, and the type 'typed', a packaging of a value with its type, suitable for use with the interpreter. The command interpreter has the same integrity as preprogrammed Ten15 in that no operation can be incorrectly applied.

The FLEX editor and command interpreter [8] together display these precepts, but in a largely untyped domain. Further study is required into their realisation within a statically typed framework, and also into the functionality of the command interpreter. In the demonstration system, the command interpreter, TRUC, is quite primitive in that it does not provide the full power of the Ten15 machine, and the type system is overridden, although in a controlled manner, when loading modules.

Work item:
consider functional power of command interpreter

5.7 *Diagnostics*

On FLEX we have a post-mortem debugger which can relate values in the workspaces of failed procedures back to source text. This tool can easily be adapted for any single-process Ten15 system. In the longer term we would like an interactive debugger, particularly when multiprocessing. Interpretation is one possible approach, but often it is desirable or even necessary to work with code more closely resembling the normal compiled output. To develop a good debugger we would have to formalise the notion of break points in Ten15 and consider carefully the consequences for data security. Useful visualisation of the confusing amount of data available in a fully parallel system will also need much work.

Work items:
write an interactive debugger
devise methods of debugging distributed Ten15

5.8 *Memory model*

The model of memory presented by Ten15 is based on the allocation, on demand, of an amount of memory sufficient to contain a value of some particular type. Such allocations deliver a pointer value that allows access to that region of store; such pointers cannot be forged, and integrity requirements mean that primitive concepts such as 'address' do not play a part.

There is no explicit return-to-free operation (it would be neither sound, complete nor desirable), so for all practical purposes some kind of garbage collector is required in order to recycle inaccessible storage. There is still the problem of what to do when there really isn't enough store available. On FLEX the requesting process is failed — this is an unsatisfactory heuristic that takes no account of how much memory any individual process uses and rarely yields useful diagnostics. (Worse still, if the process forms a vital part of the system the consequences are usually disastrous!)

Sequential Ten15 does already admit of more than one store, though to date this aspect has been treated in a somewhat ad-hoc fashion. There is a single mainstore, which is used to hold the contents of pointers, vectors or arrays, together with any working space that an implementation may need. There may also be several persistent datastores, which provide permanent disc storage. Parallel versions of Ten15 will also have to allow for the presence of multiple memories, and consider the consequences of sharing a single store between several processors. The operations on these different kinds of stores are significantly different. We need to produce coherent models of these kinds of stores, especially in a parallel system where many mainstores may exist, and consider whether these models can and should be unified.

In safety-critical and other applications it is important that memory is not dynamically allocated, or at least that it is known precisely how much heap is needed. We would like to identify a static subset of Ten15. We believe that the operations which demand heap allocation can be identified.

Work items:
- out of memory exceptions
- develop alternative memory models and consider their unification†
- static subsets of Ten15

5.9 *Mainstore*

The mainstore in Ten15 can be considered as serving two major purposes. Most obviously, the contents of pointers, vectors and arrays are kept here, allowing for mutability, aliassing, etc. In addition, it is used by an implementation as a working store, to represent values that do not formally reside in memory, such as types, and in order to realize implicit concepts such as procedure workspaces. The mainstore is not modelled as a separate store, but is assumed to be closely attached to some processor, which performs operations on it.

There is a compromise to be found between the need for dynamic management of memory and the goal of program efficiency. In particular, the garbage collection overhead should not be intrusive, especially during highly interactive activities, such as editing. The garbage collector is currently implemented as a variant of the mark/sweep algorithm, which is linear in all of its parameters; the translator ensures that the run-time representations used are such that scalars and non-scalars may be readily distinguished. A number of well-known techniques are used to reduce the amount of memory allocated, and hence the frequency of collection. Despite these efforts there is a noticeable interruption of processing during garbage collection (a second or so on FLEX), and with larger memories the interruption could easily become intolerable.

Work item:
- *consider on-the-fly garbage collection*

† Initial work on this topic may be found in [9].

5.10 Multiple and shared memories

With the introduction of parallelism into Ten15 we have to consider the consequences of sharing a single store between several processors, and allow for the presence of multiple memories. Processes can interact only by explicitly communicating some value or by causing side-effects on some data-structure that has previously been transmitted.

Within a single memory, arbitrarily complex values can be communicated with essentially infinite bandwidth by passing pointers or side-effecting. Transfers from one store to another generally involves some physical link of limited bandwidth, so there are significant efficiency differences.

Communication between distributed memories leads to an important semantic distinction, in that transferring a value in the same store results in sharing of the same value, whereas transferring to another store yields only an isomorphic copy (which preserves sharing and circularities within the value itself, but not with other values). Other significant differences can result from the transient nature of some kinds of value, or their implicit binding to one particular location. Device drivers, for example, are obviously attached to some physical entity. Because of the change of semantics involved, it may not always be appropriate to transfer such values between memory spaces. For each such kind of value, an ad-hoc decision has been taken whether to allow the transfer, with its change of meaning, or to forbid it. It is not always possible to predict whether any particular value will be transferrable, due to abstractions such as procedure non-locals, so that dynamic tests are needed.

Properties that are true on a uniprocessor can not always be assumed to hold when more than one processor shares direct access to a memory. The best sequential algorithms are often not the most appropriate techniques when several processors can access the memory simultaneously. For example, the correctness of many garbage collection algorithms depends on the store not being altered whilst they are working. Co-operative scheduling on a single individual processor is often enough to ensure atomicity of critical functions, but the store of a shared-memory machine is accessed as if the scheduling were pre-emptive.

On the other hand, there are some hard problems associated with distributed memories. Before complex values can be communicated between stores, they may have to be 'flattened' to fit through the physical links joining them, and reconstructed at the other end; this is automatically done by the Ten15 kernel. If a system is homogeneous (i.e., scalar data has the same representation everywhere), and the same translator resides on each node, then the data can be sent directly as described. With a heterogeneous network only untranslated Ten15 can be transmitted, to be translated either at the remote node or by a trusted cross-translator on the host. We need to study carefully the consequences of such heterogeneity on the representations we can use.

Furthermore, it is practically essential that there can be cross-store pointers, for otherwise interaction between processes residing in different memories is nigh impossible. However, there is then significant difficulty in garbage collection, mostly because of the need to preserve values that are only referenced from a remote store. It is also useful to be able to perform operations on values stored remotely; the most important class is to be able to invoke procedures on another machine with some kind of remote procedure call (RPC) mechanism. This can be used to maintain consistency across the system by centralising some facility at a single node.

Work items:
garbage collection of shared and distributed memory
atomic allocation and access of shared and distributed memory
parallel on-the-fly GC
transfer of untransferrable values
heterogeneous systems
cross-store pointers

5.11 Persistent datastores

Ten15 splits memory into two levels: volatile mainstore and persistent filestore (datastores). In contrast to mainstore, there is an explicit model of datastore as an entity outside the main processing unit, be it a Winchester disc, CD-ROM, magnetic tape or whatever. There is no concept of ownership of a datastore — any CPU/mainstore may talk to any number of datastores. Conversely, datastore values cannot refer to anything in mainstore, nor, at the moment, to anything on other datastores. This decoupling reflects that datastores have facets greatly different from mainstores: they can be detached, they hold much more data, but in larger granules, this data is persistent, and traffic is slow. We nevertheless wish to preserve the design intent of Ten15 in the datastore model — thus datastore values can have almost any type and their integrity cannot be corrupted. To simplify this latter goal present implementations of datastores are non-overwriting. Each value is written to a fresh area of the store, then the root reference updated atomically. This simple atomic updating model proved inadequate for database transactions, and a method of updating several disc locations together, called transactions, has been added.

Some kinds of value cannot be copied onto a persistent store because of their temporal nature; for example, a transaction cannot be persisted, then resumed at a later date. For each such kind of value, an ad-hoc decision has been taken whether to allow the transfer, with change of meaning, or to forbid it.

There are optimisations which cut down on datastore traffic. On reading a value into mainstore it is effectively cached until the next mainstore garbage collection, thus obviating any reread of the value.

Datastore garbage collection occurs off line, when the store is full. On-line datastore garbage collection is presently precluded by inadequate mainstore resources. Our current garbage collector does not perform compaction, but it would be possible to compact the datastore without corrupting the data.

Work items:
persistence of unpersistable values
cross-datastore pointers
on-line datastore garbage collection

5.12 Parallelism and networking

Ten15 V0-p has extensions for pseudoparallelism, but does not explicitly address distribution of program and data to remote processors. It is hoped that its pseudoparallel operations will prove suitable for use within a truly parallel programming environment. To map programs and data onto a parallel architecture, extra operations concerned with distribution remain to be defined and efficient implementations found.

Distribution across a network may be a special case of the implementation of Ten15 on a distributed memory multi-processor. From FLEX we have experience of transmitting data between homogeneous machines connected via Ethernet, and of using RPCs.

One extra problem is to communicate information to and from the world outside Ten15. To preserve the integrity of Ten15, data formatted outside must either be sufficiently primitive that we know it must be safe (such as a byte stream), or it has to be trusted.

Work items:

- distribution across homogeneous close-coupled networks
- unification of loose and close-coupled networks
- trustworthiness of data
- distribution across networks
- memory allocation and garbage collection for distributed Ten15

5.13 Programming languages and Ten15

The main use to date of Ten15 has been as a common target language for compilers. Algol68 and Pascal compilers exist for V0, one for Standard ML is almost complete, and one for Ada has been partially developed. Full mappings from Ada to V1 have also been produced. Whilst integrity constraints mean that unsafe languages such as C are not suitable sources, it ought in principle to be possible to compile any strongly typed language to Ten15. However, a few language features, such as the first-class continuations found in Scheme, are not well supported and may cause implementations to be rather inefficient. An important part of our research will be to evolve so that such features can be realistically provided.

On the other hand, it is not possible to obtain the full power of Ten15 by compilation from existing languages. An 'assembler' is required so that we can create any specific fragment of Ten15, which can then be combined with the output of other compilers. The notation that originally produced these small fragments has since evolved into a substantial programming language in its own right. The success of Ten15 must surely be linked with that of a language, or possibly a family of languages, that are capable of harnessing its full power. So those languages must now be considered an important part of the Ten15 world, and had better be good.

Unfortunately, as well as gaining syntactic power, the present notation has garnered a large range of non-orthogonal historic encrustations which must be cleaned away. We started producing a new compiler which reformed the worst of these, like the module system. Strict upward compatibility was not required, but we did wish to salvage as much existing code as possible. Excepting system problems, this new compiler is largely complete. For the future we will have to consider the functionality, style and power we want from new languages.

Work items:

- evolution to extend scope of language support
- complete new notation compiler
- consider expressiveness of a new language

6 Conclusion

Ten15 addresses a large number of problems in computing, particularly in the fields of data integrity and systems programming, which are outside the remit of TDF. Nevertheless, in many respects its structure and functionality is inadequate. Given sufficient time and resources Ten15 could no doubt have been extended to cover a much broader spectrum of programming styles and architectures, and to do so with greater ease and assurance than it presently does.

The review of the inadequacies and suggested extensions presented in this paper is wide ranging, and the effort it implies is very high. The language needs to be considered as a whole; piecemeal extensions, for example the facilities for parallelism added by the COOTS team, occlude the real goals of the project. Furthermore, technical problems were being encountered in building the demonstration system. Consequent upon the lack of necessary resources the development of Ten15 along that path was abandoned.

We believe that future work programmes will adopt the ideas of Ten15. In this document, we have discussed weaknesses in its definition, and consequences, both positive and negative, resulting from its all-embracing approach. Recognition of these aspects will help future teams to identify specific problems for investigation. To this extent, at least, the Ten15 experience has been valuable.

Acknowledgement

The authors acknowledge the assistance of P W Edwards and I F Currie of the Computer Systems Engineering Division of RSRE in drafting the original COOTS working paper upon which the bulk of this memorandum is based.

References

1 "The Algebraic Specification of a Target Machine: Ten15"
J M Foster
Published in "High-Integrity Software", ed. C T Sennett
Pitman, 1989

2 "Ten15 Developments to Support Parallelism"
P W Edwards, D J Tombs, D I Bruce
IED 3/1/1059 COOTS deliverable 2.1, June 1991
Republished as RSRE Memorandum 4545, November 1991

3 "TDF Specification"
J M Foster, M Brandreth, P W Core, I F Currie, N E Peeling
RSRE Report 91005, October 1990

4 "Ten15 Prototype"
J M Foster, I F Currie, N E Peeling, PW Edwards, M Stanley, P W Core, M Brandreth
RSRE Report 91025, November 1991

5 "The Ten15 Signature: The Definition of Ten15 — Version 0"
D I Bruce
RSRE internal documentation, March 1990

6 "PerqFlex Firmware"
I F Currie, J M Foster, P W Edwards
RSRE Report 85015, December1985

7 "A Syntax Improving Program"
J M Foster
Computer Journal, Vol. 11, No. 1, May 1968, pp 31–34

8 "An Evaluation of the FLEX Programming Support Environment"
M Stanley
RSRE Report 86003, August 1986

9 "A Strongly-Typed Approach to Parallel Systems"
D I Bruce
Proceedings of Workshop on Abstract Machine Models for Highly-Parallel Computers
Vol. 2, pp 57–59
BCS Parallel Processing Specialist Group, Leeds, March 1991

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
MEMO 4543	NOVEMBER	1991

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
THE EVOLUTION OF Ten15

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period
Project Number	Other References

Authors	Pagination and Ref
TOMBS. D J. BRUCE. D I	18

Abstract

Recently development of the RSRE intermediate language Ten15 and its associated demonstration environment was suspended owing to a shortage of resources. This memorandum describes areas for study and potential evolution for future projects having similar goals.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

888-48

INTENTIONALLY BLANK