PDV-E 122
Dezember 1978

PDV-Entwicklungsnotizen

Das Sprachentwicklungsprojekt des US-Verteidigungsministeriums

P. Elzer
Dornier-System GmbH
Friedrichshafen

Kernforschungszentrum Karlsruhe

PDV-Berichte

Die Kernforschungszentrum Karlsruhe GmbH koordiniert und betreut im Auftrag des Bundesministers für Forschung und Technologie das im Rahmen der Datenverarbeitungsprogramme der Bundesregierung geförderte Projekt Prozeßlenkung mit Datenverarbeitungsanlagen (PDV). Hierbei arbeitet sie eng mit Unternehmen der gewerblichen Wirtschaft und Einrichtungen der öffentlichen Hand zusammen. Als Projektträger gibt sie die Schriftenreihe PDV-Berichte heraus. Darin werden Entwicklungsunterlagen zur Verfügung gestellt, die einer raschen und breiteren Anwendung der Datenverarbeitung in der Prozeßlenkung dienen sollen.

Der vorliegende Bericht dokumentiert Kenntnisse und Ergebnisse, die im Projekt PDV gewonnen wurden.

Druck und Verbreitung:

Kernforschungszentrum Karlsruhe GmbH
Postfach 3640 7500 Karlsruhe 1

Bundesrepublik Deutschland

PDV-E 122

PROJEKT PROZESSLENKUNG MIT DV-ANLAGEN
ENTWICKLUNGSNOTIZ PDV-E 122

DAS SPRACHENTWICKLUNGSPROJEKT DES US-VERTEIDIGUNGSMINISTERIUMS

P. ELZER

DORNIER-SYSTEM GMBH

FRIEDRICHSHAFEN

152 SEITEN

DEZEMBER 1978

Vorwort

Die Entwicklung einer neuen höheren Programmiersprache für integrierte Rechnersysteme (embedded computer systems) im US-Verteidigungsministerium findet große Aufmerksamkeit in der internationalen "computing community". Diese Aufmerksamkeit wird hervorgerufen durch verschiedene Aspekte des Sprachentwicklungsprogramms:

1. Die Sprachentwicklung ist von großem wissenschaftlichen Interesse, da international anerkannte Fachleute daran mitarbeiten.

2. Die Organisation und Zeitplanung des Sprachentwicklungsprojektes ist aufsehenerregend, da bisher alle Termine eingehalten wurden. Dies ist erstaunlich, da die Terminvorgaben nach allgemeinen Erfahrungen bei derartigen Projekten sehr knapp bemessen sind.

3. Gelingt es tatsächlich, die entstehende Sprache verbindlich für alle Projekte des US-Verteidigungsministeriums vorzuschreiben, so hat dies auf Grund der politischen Lage zur Folge, daß die Sprache primär im Verteidigungsbereich und sekundär im industriellen Bereich auch in Deutschland wirtschaftlich bedeutsam werden könnte. Um die mögliche wirtschaftliche Bedeutung dieser Sprachentwicklung für die Bundesrepublik abzuschätzen und einen intensiven Informationsfluß zwischen beiden Seiten herzustellen, wurde Herr Elzer, Firma Dornier-System GmbH, Friedrichshafen, im Rahmen des PDV-Projekts im Auftrag des BMFT für ca. 1 Jahr nach USA delegiert. Herr Elzer ist dort Mitarbeiter in der Gruppe, die das Sprachentwurfsprojekt koordiniert.

Als weitere Publikation über die Ziele und den Verlauf des Sprachprojekts ist ein Artikel von W.A. Whitaker, dem Leiter dieser Sprachgruppe, zu nennen /1/. Es soll nicht unerwähnt bleiben, daß an der Vorgehensweise beim Sprachentwurf und an der Qualität der bisher vorliegenden Sprachentwürfe heftige Kritik geübt wurde /2/, /3/. Eine Abgrenzung der Anwendungsgebiete für die neu entwickelte Sprache des US-Verteidigungsministeriums und für PEARL wird in /4/ gegeben.

Literatur:

/1/ W.A. Whitaker: The U.S. Department of Defense Common High Order Language Effort. SIGPLAN Notices, Vol. 13 (1978), Nr. 2, S. 19-29.

/2/ E.W. Dijkstra: DOD-I: The Summing Up. SIGPLAN Notices, Vol. 13 (1978), Nr. 7, S. 21-26.

/3/ E.W. Dijkstra: On the BLUE/GREEN/YELLOW/RED Language submitted to the DOD. SIGPLAN Notices, Vol. 13 (1978), Nr. 10, S. 10-32.

/4/ T. Martin: Neue amerikanische Programmiersprache am Horizont. Regelungstechnische Praxis, 20 (1978), Nr. 11.

KURZFASSUNG

Dieser Bericht gibt eine gedrängte Darstellung von Geschichte, Zielen und Organisationsstruktur des Projektes zur Schaffung einer gemeinsamen höheren Programmiersprache für integrierte Rechnersysteme des US-Verteidigungsministeriums. Es wird der Stand der Arbeiten an Sprache und Softwareumgebung im September 1978 beschrieben, sowie eine Reihe von begleitenden Aktivitäten geschildert und ein Vergleich mit PEARL versucht.

ABSTRACT

This report gives a condensed presentation of history, aims and organisational structure of the US-DoD project for a Common High Order Language for embedded computer systems. It describes the state of the work on language and software-environment in September 1978, as well as some supporting activities. A comparison with PEARL is attempted.

INHALTSVERZEICHNIS

		Seite
0	**Einleitung**	4
1	**Das eigentliche Sprachentwicklungsprojekt**	5
1.1	Vorgeschichte und organisatorische Grundlagen	5
1.2	Entwicklung der technischen Anforderungen	7
1.3	Untersuchung von Kandidatensprachen	9
1.4	Die "Interim List"	11
1.5	Der bisherige Verlauf der Sprachentwicklung	12
1.6	Weitere Planungen	13
2	**Begleitende Aktivitäten**	14
2.1	Die Softwareumgebung	14
2.2	'TARTAN' - ein Sprachmodell	16
2.3	Testprobleme	17
2.4	Untersuchungen zur Ein-/Ausgabe	18
2.5	Implementationsvorbereitungen	19
2.6	Das ARPA-Netz	20
3	**Die Sprachentwicklung des US-Dept. of Def. im Vergleich mit PEARL**	21
3.1	Zielsetzung	21
3.2	Technische Gesichtspunkte	23
4	**Literatur**	25

Anhänge:

A1 D.A. Fisher:
DoD's Common Programming Language Effort

A2 Zusammenfassung von Wirtschaftlichkeitsanalysen

A3 US-Dept. of Def. Requirements for High Order Computer Programming Languages ("STEELMAN")

A4 The US-Department of Defense Common High Order Language Effort

A5 Department of Defense Requirements for the Programming Environment for the Common High Order Language ("PEBBLEMAN")

A6 Shaw, Hilfinger, Wulf:
TARTAN, Language Design for the Ironman Requirement: Reference Manual

A7 Shaw Hilfinger, Wulf:
TARTAN, Language Design for the Ironman Requirement: Notes and Examples

A8 Set of Sample Problems for Phase II of the design contracts of the Dod HOL commonality effort

0. EINLEITUNG

Im Jahre 1975 wurde vom US-Verteidigungsministerium ein Projekt begonnen mit dem Ziel, für den Verteidigungsbereich eine einheitliche höhere Programmiersprache für integrierte Rechnersysteme(1) zu schaffen. Dabei wurde von Anfang an angestrebt, auch die für einen rationellen Einsatz dieser Sprache notwendigen Softwarehilfsmittel, wie Computer, Laufzeitpakete, Betriebssystemergänzungen, Testhilfen, etc. in der Sprache selbst erstellen zu können, um maximale Portabilität zu erzielen. Parallel zur eigentlichen Sprachentwicklung wird deshalb auch die notwendige Softwareumgebung mit definiert. Außerdem sind Organisationen zur Pflege der Sprache, Überwachung ihrer Implementierungen und Unterstützung ihrer Anwendungen geplant. Ein Projektziel ist es, bis 1980 die endgültige Sprachspezifikation, sowie mindestens einen Produktionscompiler zur Verfügung zu haben.

Im vorliegenden Bericht wird versucht, einen knappen Gesamtüberblick über die Geschichte des Projektes, seine Ziele und seine Organisationsstruktur zu geben, sowie andere relevant erscheinende Aktivitäten, die nicht immer organisatorisch mit dem Sprachentwicklungsprojekt verknüpft sein müssen, zu identifizieren. Wenn auch vor Veröffentlichung der endgültigen Sprachvorschläge im Frühjahr 1979 keine exakten Aussagen über Charakter und Funktion von Elementen der Sprache gemacht werden können, so wird doch versucht, die bereits an Hand der veröffentlichten technischen Anforderungen erkennbaren Unterschiede zu und Ähnlichkeiten mit PEARL zu charakterisieren.

Für den Leser, der an eingehender Information interessiert ist, sind die wesentlichsten Originaldarstellungen als Anhänge beigefügt.

(1) dies ist der Versuch einer Übersetzung des amerikanischen Fachbegriffs "embedded computer system", der nicht vollständig dem deutschen Begriff des "Realzeitsystems" entspricht.

1. DAS EIGENTLICHE SPRACHENTWICKLUNGSPROJEKT

1.1 Vorgeschichte und organisatorische Grundlagen

In diesem Abschnitt soll nur ein zusammenfassender Überblick gegeben werden, da eine sehr eingehende Gesamtdarstellung der Geschichte des Projektes, seiner Ziele und seiner Organisationsstruktur bereits im April 1978 in der Zeitschrift 'Computer' der IEEE erschienen ist. Dieser Aufsatz ist als Anhang 1 beigefügt.

Bereits in den Jahren 1973 und 1974 wurden Studien durchgeführt [1,2] um Daten über Höhe und Verteilung der Softwarekosten im US-Verteidigungsbereich zu erhalten.

Zwei Ergebnisse waren besonders wesentlich:

- Die Kosten für integrierte Rechnersysteme stellten mit 56 % den Hauptanteil der jährlichen Ausgaben von 3 Mrd. Dollar für Software im US-Verteidigungsbereich dar.

- Bei einer Berechnung über die gesamte Lebensdauer eines Systems übertrafen die Kosten für die Wartung die für die Entwicklung und Herstellung bei weitem.

Weiterhin stellte sich heraus, daß im gesamten Verteidigungsbereich über 200 Rechnermodelle und über 450 verschiedene Programmiersprachen (einschließlich Assembler) verwendet wurden. Aus dieser Zersplitterung ergab sich weiter, daß die für eine rationelle Softwareentwicklung und -wartung notwendigen Hilfsmittel nur in den seltensten Fällen und dann auch meist nur in rudimentärer Form vorhanden waren. Eine Schlüsselrolle spielte auch hier das Fehlen einer einheitlichen Programmiersprache.

Aus diesen Gründen wurde im Januar 1975 auf Initiative des "Director of Defense, Research and Engineering"(2) ein gemeinsames Programm der Teilstreitkräfte formuliert. Außerdem wurden keine weiteren Mittel für Entwicklung und Einsatz neuer Programmiersprachen in wesentlichen Projekten des Verteidigungsbereiches mehr bereitgestellt, bis das Problem einer zufriedenstellenden gemeinsamen Nutzung softwarebezogener Hilfsmittel gelöst wäre.

(2) heute: Undersecretary of Defense, Research and Engineering"

Ungefähr zur selben Zeit wurden andere Programme gestartet, z.B. zur Untersuchung der Möglichkeit einer einheitlichen Rechnerfamilie für den Verteidigungsbereich (MCF: "military computer family").

Globale Richtlinien für die Behandlung der Probleme rationelleren Rechnereinsatzes gab die Anweisung 5000.29 [3]. Eine Zusammenstellung verschiedener Artikel über die erkannten Probleme und Vorschläge zu ihrer Behebung erschien im Oktoberheft 1975 des "Defense Management Journal" [4].

Um die Arbeiten an der Sprachentwicklung zu koordinieren, wurde im Januar 1975 die "High Order Language Working Group (= HOLWG)" gegründet. Stimmberechtigte Mitglieder sind Vertreter der US-Armee, Marine, Marineinfanterie und Luftwaffe, sowie des Amtes für Nachrichtenwesen, des Nationalen Sicherheitsbüros und der DARPA(3). Der Vorsitzende der HOLWG, z.Zt. Lt. Col. W.A. Whitaker, wird vom USDRE ernannt. Dr. D. Fisher vom "Institute for Defense Analyses (IDA)" ist der technische Berater dieses Ausschusses. Die HOLWG ist gegenüber dem USDRE verantwortlich und außerdem tätig als einer der Unterausschüsse des "Management Steering Committee for Embedded Computer Resources (MSC–ECR)".

Ihre Aufgabe ist es, "Möglichkeiten zur Einführung der minimalen Anzahl gemeinsamer höherer Programmiersprachen zu untersuchen, die bei Entwicklung, Beschaffung und Betrieb von Rechnern in militärischen Systemen verwendet werden sollen". Insbesondere sollen die technischen Anforderungen an derartige Sprachen definiert, die Eignung von existierenden Sprachen untersucht, eine erfolgsversprechende Vorgehensweise festgelegt, und die notwendigen Maßnahmen überwacht werden.

Der mögliche Nutzen des Sprachentwicklungsprojektes wurde in mehreren Wirtschaftlichkeitsanalysen untersucht, die zu positiven Ergebnissen führten. Anhang 2 enthält die Zusammenfassungen von zweien dieser Analysen.

(3) "Defense advanced research projects agency", eine zentrale Forschungsförderungsstelle des US-Verteidigungsministeriums.

1.2 Entwicklung der technischen Anforderungen

Im Gegensatz zum Vorgehen bei verschiedenen anderen Sprachentwicklungsprojekten betätigt sich die HOLWG nicht als Sprachentwicklungsausschuß im üblichen Sinne, sondern fördert die Entwicklung durch unabhängige Auftragnehmer auf Wettbewerbsbasis.

Als Arbeitsgrundlage wurden die technischen Anforderungen an eine höhere Sprache zur Programmierung von integrierten Rechnersystemen zusammengestellt. Dieses Dokument diente dann später als Grundlage für eine Ausschreibung, lieferte einen allgemein anerkannten Bewertungsmaßstab und verhinderte, daß völlig unvergleichbare Sprachentwürfe entstanden.

Die gegenwärtigen technischen Anforderungen, die in dem als Anhang 3 beigefügten "Steelman"-Papier zusammengestellt sind, sind das Ergebnis eines evolutionären Prozesses, der von 1975 bis 1978 dauerte.

Am IDA wurde eine Serie von Vorschlägen ausgearbeitet, wobei jeweils die Kommentare zum vorhergehenden Vorschlag eingearbeitet wurden, die seitens potentieller Benutzer, Auftragnehmern des Militärbereichs und anderer interessierter Organisationen eingingen. Auf diese Weise konnten die einschlägigen Erfahrungen eines großen Teiles der Fachwelt genutzt werden. In gewissen Zeitabständen wurden einzelne Versionen durch militärische Dienststellen offiziell genehmigt und als die folgenden Dokumente veröffentlicht:

STRAWMAN	April 1975
WOODENMAN	August 1975
TINMAN	Januar 1976
IRONMAN	Januar 1977
revised IRONMAN	Juli 1977
STEELMAN	Juni 1978

Als ein Beitrag zur Diskussion um TINMAN wurde im Herbst 1976 ein Seminar mit anerkannten Wissenschaftlern auf den Gebieten Sprachentwurf und Compilerbau an der Cornell-Universität in Ithaca, N.Y., veranstaltet. Die Ergebnisse dieses Seminars sind in Buchform erhältlich [5].

Insgesamt gingen während der Entwicklung der technischen Anforderungen über 2000 Seiten an Kommentaren von 184 Institutionen und Einzelpersonen ein.

Die Entwicklung der technischen Anforderungen erbrachte aber noch ein weiteres wesentliches Ergebnis: Es wurde festgestellt, daß in allen Bereichen des Verteidigungssektors die gleichen Anforderungen an eine höhere Programmiersprache für integrierte Systeme galten. Dieses Ergebnis war als nicht selbstverständlich vorauszusetzen gewesen.

1.3 Untersuchung von Kandidatensprachen

Da nicht von vornherein feststand, daß eine neue Sprache entwickelt werden mußte, wurden im Laufe des Jahres 1976 23 existierende Programmiersprachen daraufhin untersucht, wie weit sie den aufgestellten technischen Anforderungen genügten. Diese Vergleiche wurden nach von der HOLWG aufgestellten Richtlinien von insgesamt 16 Auftragnehmern durchgeführt.

Folgende Sprachen wurden betrachtet:

ALGOL 60
ALGOL 68
CMS-2
COBOL
CORAL 66
CS-4
EL-1
EUCLID
FORTRAN
HAL/S
J 3 B
J 73
LIS
LTR
MORAL
PASCAL
PDL 2
PEARL
PL/I
RTL/2
SIMULA 67
SPL/1
TACPOL

Jede Sprache wurde von mindestens zwei Bewertungsgruppen begutachtet, die Ergebnisse von einem Ausschuss aufbereitet und der HOLWG vorgelegt. Das gesamte während der Auswertung entstandene Material ist auf Mikrofilm erhältlich [6].

Wesentliche Ergebnisse waren:

- Keine der Kandidatensprachen erfüllte die technischen Anforderungen in ausreichendem Maß, um als "die" endgültige einheitliche Programmiersprache akzeptiert werden zu können.

- Aus der Tatsache, daß mehrere der Kandidatensprachen bereits eine Anzahl der aufgestellten Anforderungen erfüllten, ergab sich, daß das Problem im Rahmen des Standes der Technik lösbar war, d.h. ein geeigneter Sprachentwurf erschien machbar.

- Eine vollständige Neuentwicklung erschien deshalb nicht als notwendig, weil alle diejenigen Sprachen, die die technischen Anforderungen zu einem großen Teil erfüllten, abgeleitet waren von ALGOL 68, PASCAL oder PL/I. Diese drei Sprachen wurden deshalb als Basissprachen für die Entwicklung der gemeinsamen Programmiersprache vorgeschlagen.

1.4 Die Interim List

In der Zwischenzeit wurde auch ein Vorhaben verwirklicht, von dem man sich eine wesentliche Verbesserung der Situation schon vor der Durchsetzung einer gemeinsamen Programmiersprache versprach. Es wurde eine Liste von einigen wenigen Programmiersprachen zusammengestellt, deren Verwendung für neu zu beginnende Projekte integrierter Rechnersysteme vorgeschrieben werden konnte.

Kriterien für die Auswahl dieser Sprachen waren vor allem, daß sie

- ihre Eignung durch praktischen Einsatz bei mindestens einem der Armeeteile bewiesen haben
  und
- durch ein geeignetes Definitionsdokument festgelegt sein

mußten. Aufnahme in die aufzustellende Liste setzte außerdem die Verpflichtung zur weiteren Betreuung der Sprache seitens der sie nominierenden Organisation voraus.

Diese Voraussetzungen waren bei folgenden, im VS-Verteidigungsbereich bereits seit einiger Zeit verwendeten Sprachen erfüllt:

CMS-2
SPL-1
TACPOL
J 3 JOVIAL
J 73 JOVIAL
ANSI COBOL
ANSI FORTRAN

Diese Sprachen wurden in einer Anweisung 5000.31 [7] als verbindlich für den Einsatz bei neu zu beginnenen Projekten erklärt, "falls nicht nachgewiesen werden kann, daß die Verwendung einer anderen Sprache über die Lebensdauer des Systems gesehen kostenwirksamer ist" (cit. 5000.31).

1.5 Der bisherige Verlauf der Sprachentwicklung

Nach der Formulierung eines Projektplanes wurde dann im April 1977 die Sprachentwicklung international ausgeschrieben. Von den 18 eingegangenen Angeboten wurden folgende vier ausgewählt:

CII-Honeywell-Bull (Paris, Minneapolis)
Intermetrics (Nähe Boston)
Softech (Nähe Boston)
Stanford Research International (Nähe San Francisco).

Alle vier erfolgreichen Anbieter schlugen eine Sprachentwicklung auf der Basis von PASCAL vor. Die Arbeiten begannen im August 1977 und die Ergebnisse der ersten Phase, die vorläufigen Sprachentwürfe, wurden termingerecht im Februar 1978 ausgeliefert.

Diese Sprachentwürfe wurden dann einem Bewertungsverfahren unterzogen, an dem über 70 Teams und Einzelpersonen aus aller Welt teilnahmen. Um die Urheberschaft der einzelnen Sprachen geheimhalten zu können, wurden die Sprachen durch einen Farbcode (grün, rot, blau, gelb) identifiziert. Die Ergebnisse dieser Einzelbewertungen wurden durch eine Expertengruppe ausgewertet und mit Empfehlungen der HOLWG zur Entscheidung vorgelegt. Es wurde beschlossen, die Sprachentwürfe "grün" (CII-Honeywell-Bull) und "rot" (Intermetrics) weiterentwickeln zu lassen.

Eine Kurzdarstellung des Auswerteverfahrens mit statistischen Daten findet sich in [8]. Eine Zusammenfassung des Projektverlaufes bis zur Vergabe der Entwicklungsaufträge gibt ein Vortrag von Lt. Col. W.A. Whitaker, dessen Manuskript als Anhang 4 beigefügt ist. Ein Gesamtbericht über die Auswertung ist auf Mikrofilm erhältlich [9]. Er enthält folgendes Material:
Anleitung zur Durchführung der Analyse"revised IRONMAN, alle vier Sprachvorschläge der Phase I, alle Einzelanalysen, die nach Sachgebieten umgeordneten Analysen, und das "STEELMAN"-Dokument.

Anschließend an diese Auswertungsphase begann im April 1978 die zweite Phase der Sprachentwicklung, die Feindefinition, die mit der Vorlage der vorläufigen vollständigen Sprachbeschreibung im März 1979 beendet sein soll.

1.6 Weitere Planungen

Während der zweiten Phase der Sprachentwicklung werden durch Vertreter der HOLWG in vierteljährlichen Abständen Fortschrittskontrollen bei den Auftragnehmern durchgeführt, deren Ergebnisse allerdings aus wettbewerbsrechtlichen Gründen vertraulich behandelt werden müssen. Bei der zweiten derartigen Veranstaltung im November 1978 werden auch Vertreter ausgewählter Bewertungsteams hinzugezogen werden.

Im April 1979 soll dann die Entscheidung für eine der beiden in der zweiten Phase entwickelten Sprachen fallen. Daran wird sich bis Dezember 1979 eine Phase eingehender Tests anschließen, während der auch noch evtl. notwendige Verfeinerungen vorgenommen werden sollen. Auch an dieser Phase sollen ausgewählte internationale Teams beteiligt werden. Zur Unterstützung dieser Arbeiten sollen im April 1979 auch Testübersetzer für die in der zweiten Phase entwickelten Sprachen auf dem Arpanetz verfügbar sein.

Gleichzeitig soll mit der Erstellung der ersten Produktionscompiler begonnen werden, und es wird erwartet, daß bis Mitte 1980 zumindest einer zur Verfügung stehen wird.

Im Oktober 1978 wird mit der Vorbereitung von Kursmaterial begonnen werden, um den Beginn von Ausbildungskursen im Frühjahr 1979 zu ermöglichen. Auch die organisatorischen Vorbereitungen für eine Betreuungsstelle werden schon im Herbst 1978 anlaufen, damit diese Stelle im Frühjahr 1980 ihre Arbeit aufnehmen kann.

2. BEGLEITENDE AKTIVITÄTEN

2.1 Die Softwareumgebung

Das Sprachentwicklungsprojekt sollte von vornherein nicht als isolierte Aktivität gesehen werden, sondern eingebettet in allgemeine Bemühungen, die Softwareproduktion auf dem US-Verteidigungssektor insgesamt zu konsolidieren und damit schließlich zu rationalisieren und zu verbilligen. Es war deshalb von Anfang an beabsichtigt, die Anwendung der Sprache durch entsprechende Softwarewerkzeuge zu unterstützen. Außerdem sollten Organisationen geschaffen werden, die sowohl Kontrolle über die Sprache selbst ausüben, als auch eine Qualitätsprüfung der erstellten Compiler vornehmen und Anwenderberatung vornehmen könnten. Entsprechende Maßnahmen sind deshalb bereits in den technischen Anforderungen an die Sprache angedeutet (vergl. Kap. 13 des "STEELMAN"-Dokuments).

Es ist nun beabsichtigt, eine Reihe von Dokumenten zu entwickeln, die ähnlich wie die STRAWMAN-STEELMAN-Serie eine Reihe immer weiter konsolidierter Anforderungen an Organisationen und Softwareumgebung beschreiben sollen.

Als Vorbereitung dazu begannen bereits 1977 Arbeiten an der Definition der Anforderungen an Betreuungsorganisationen und Softwareumgebung. Zwei Unterauftragnehmer fertigten unter der technischen Aufsicht eines Vertreters der US-Marine Vorstudien an, die im Januar, bzw. April 1978 ausgeliefert wurden. Diese Studien dienten als Basis für die Erstellung eines Dokumentes, das, ähnlich wie 'STRAWMAN' für die Sprache, die Diskussion über die Eigenschaften der Softwareumgebung auf breiter Basis eröffnen soll.

Nachdem aber auf diesem Gebiet an mehreren Stellen Neuland betreten werden mußte und sollte, erschien es nützlich und notwendig, gleich zu Beginn im Rahmen eines Workshop die Meinung von Fachleuten zu den Themen Softwarewerkzeuge und Sprachbetreuung einzuholen. Dieses wurde im Juni 1978 in Irvine, Universität von Kalifornien, abgehalten. Die Ergebnisse sollen noch im Laufe des Jahres 1978 veröffentlicht werden.

Im folgenden soll eine sehr kurze und deshalb vielleicht etwas schlagwortartige Zusammenfassung der als besonders wichtig erkannten Problemgebiete gegeben werden.

- Die größten Schwierigkeiten treten bei Entwurf und Wartung von Software auf

- Die Wartung und laufende Anpassung verschlingt einen noch höheren Anteil der Lebensdauerkosten eines Systems als bisher schon angenommen, nämlich bis zu 95 %.

- Die im Verteidigungsbereich (speziell bei Wartungsstellen) und bei forschungsorientierten Institutionen jeweils angewandte Softwaretechnologie klafft um Jahre, wenn nicht um eine Generation auseinander. Dagegen unterscheiden sich die zu lösenden Probleme kaum in ihrer Komplexität.

- Man verspricht sich sehr viel von Rechnerunterstützung bei Problemanalyse, Programmerstellung, und Test. Allerdings erfordern die bisher erprobten Methoden erhebliche Rechnerkapazität.

- Programmverifikation auf formaler Basis hat noch nicht den technischen Stand erreicht, der ihren praktischen Einsatz auf breiter Basis ermöglichen würde.

- Verifikation und Test von Compilern werden von den bisher damit befaßten Dienststellen nach ganz verschiedenen Methoden durchgeführt, die äußerst stark von den jeweiligen politischen Gegebenheiten abhängen. Keine Methode hat bisher alle Anforderungen erfüllen können.

- Eine Benutzerorganisation ist notwendig.

Der Hauptnutzen des Workshop für die laufende Arbeit bestand jedoch in den zahlreichen Detailbemerkungen zu den einzelnen Kapiteln des ursprünglichen Papieres zur Softwareumgebung. Dieses wurde daraufhin nach längeren Diskussionen vollständig überarbeitet und neu gegliedert.

Erste Bemerkungen, die seitens der HOLWG und über das ARPA-Netz (siehe 2.6) eingingen, wurden eingearbeitet und das resultierende Dokument, "Pebbleman", im Juli 1978 zur Diskussion und Kritik versandt. Es ist als Anhang 5 beigefügt.

Zur Zeit werden Möglichkeiten zur Verwirklichung der darin skizzierten Konzepte und die organisatorischen Voraussetzungen für die Einrichtung der erwähnten Betreuungsorganisationen geprüft.

Es ist beabsichtigt, zur Jahreswende 1978/79 eine zweite Version dieses Dokumentes zu erstellen. Erste Modellimplementationen notwendiger Softwarewerkzeuge sind ab Mitte 1979 geplant.

2.2 'TARTAN' – ein Sprachmodell

Da einer der wesentlichsten Kritikpunkte bei der Auswertung der Sprachentwürfe aus Stufe I ihre Komplexität und ihr Umfang gewesen waren, wurde im Auftrag der DARPA am "Department of Computer Science" der Carnegie-Mellon Universität in Pittsburgh eine Studie durchgeführt, mit dem Ziel, zu prüfen, mit welchem Minimalaufwand sich die Forderungen von "revised" IRONMAN erfüllen ließen.

Das Ergebnis dieser Studie war das im Juni 1978 veröffentlichte Sprachmodell 'TARTAN'. Das "reference manual", das 22 Seiten (!) umfaßt und ein Heft mit Bemerkungen und Beispielen sind als Anhänge 6 und 7 beigefügt.

Die Verfasser dieses Sprachentwurfs konnten auf die Sprachvorschläge aus Phase I, sowie auf eigene Erfahrungen mit der Entwicklung von ALPHARD [10,11] und BLISS aufbauen. Das Hauptgewicht beim Entwurf wurde auf das Typkonzept, generierende Definitionen und das Modulkonzept gelegt.

Die Ergebnisse der Studie wurden den beiden Auftragnehmern der Phase II zur Verfügung gestellt.

2.3 Testprobleme

Im Juni 1978 wurde den Auftragnehmern der Phase II ein Satz Beispielprogramme zugestellt, an Hand derer die Flexibilität und Problemgerechtheit der entstehenden Programmiersprache demonstriert werden sollen. Die Beispiele sind hauptsächlich dem Bereich der Systemprogrammierung entnommen. Es ist nicht beabsichtigt, die entstehenden Programme irgendwelchen statistischen Auswerteverfahren zu unterwerfen. Die vollständigen Testprobleme sind in Anhang 8 enthalten.

Folgende Beispiele wurden ausgewählt:

1 Erkennung asynchroner Unterbrechungen durch Abfrage
2 Unterbrechungsbehandlung unter Prioritäten
3 Ein kleines Dateibehandlungspaket
4 Darstellung bewegter Bilder
5 Ein Schutzmodul für eine Datenbasis
6 Ein Beispiel aus der Prozessteuerung
7 Adaptiver Wegeschaltalgorhythmus für einen Datenübertragungsknoten
8 Allgemeiner Zeitsteuerungsmodul
9 Parallele Ausgabe im verteilten System
10 Entpacken und Konvertieren von Eingabedaten.

2.4 Untersuchungen zur Ein-/Ausgabe

Unter der fachlichen Aufsicht eines Mitarbeiters des "Electronics Command" (Fort Monmouth) der US-Armee wurden im Rahmen einer Doktorarbeit Möglichkeiten zur Klassifizierung maschinenunabhängiger Primitivfunktionen der Ein-/Ausgabe untersucht [12]. Grundlage waren einschlägige Arbeiten von Wirth (MODULA) und Hoare, sowie die Sprachentwürfe aus Phase I.

Es wird versucht, verschiedene Ein-/Ausgabevorgänge nach ihrer inneren Funktion zu klassifizieren, etwa "Status–" oder "Unterbrechungsgesteuert" mit den jeweiligen Unterklassen. Die für die einzelnen Klassen relevanten Operationen und Steuerparameter werden identifiziert, und dazugehörige Betriebssystemtechniken und -bausteine untersucht. Außerdem werden Methoden zur Abbildung von Datenstrukturen auf Maschinendarstellung betrachtet.

Die Arbeit kann wohl am besten als der Versuch charakterisiert werden, die Implementation von E/A-Funktionen so durchzustrukturieren, daß höhere Funktionen (wie z.B. die in PEARL) in maschinenunabhängiger Weise auf die in der fertigen DoD-Sprache vorgesehenen Primitivfunktionen abgebildet werden können.

2.5 Implementationsvorbereitungen

Nachdem bereits einige Dienststellen innerhalb des Verteidigungsbereiches an einer Verwendung der zukünftigen gemeinsamen Sprache interessiert sind, werden Voruntersuchungen betrieben, durch die festgestellt werden soll, an welchen Stellen möglicherweise Implementationsschwierigkeiten zu erwarten und welche Technologien im Einzelfall am erfolgversprechendsten sind.

So wird z.B. die Verwendbarkeit von "secure UNIX" als unterliegendem Betriebssystem geprüft. Verschiedene Zwischensprachtypen werden auf ihre Eignung hin untersucht, als Grundlage für einen standardisierten, portablen Compiler zu dienen. Hierbei spielen besonders Effizienz- und Optimierungsaspekte eine Rolle. Unter diesem Gesichtspunkt müssen auch die Forschungen an der Carnegie-Mellon Universität gesehen werden, die sich z.B. mit Messungen der statischen Codeeffizienz von Compilern und mit der Entwicklung eines generierbaren, maschinenunabhängigen, portablen Compilers befassen. Um verbessertes statistisches Material zu dem Verfahren über Effizienzmessungen zu gewinnen, soll es in nächster Zeit auf existierende Compiler im Verteidigungsbereich angewandt werden. Außerdem wurde eine spezielle Untersuchung über Fragen der Softwarewartung begonnen.

2.6 Das ARPA - Netz

Dieser mehrfach erwähnte Begriff steht für ein Rechnernetzwerk von beträchtlichen Ausmaßen, das in den Jahren 1969 bis 1972 unter Förderung der ARPA aufgebaut und seitdem für eine Vielzahl von Forschungs- und Entwicklungsvorhaben benutzt wurde. Es umfaßt über 125 Rechner, an mehr als 67 über die ganzen USA verteilten Stellen, die untereinander durch kommerzielle Telefonleitungen verbunden sind. Über Wahlleitungen und Datensammelstationen kann praktisch eine unbegrenzte Anzahl von schreibenden Terminals angeschlossen werden. Die durchschnittliche Belastung während der normalen Arbeitszeit schwankt zwischen 500 und 700 Benutzern. Über Satellitenverbindungen sind Außenstationen in Europa und Hawaii angeschlossen.

Auf diesem Rechnernetz steht eine breite Palette von Softwarewerkzeugen vom Simulator für CPU s über Compiler für praktisch alle wesentlichen Programmiersprachen bis in zum Nachrichtenübermittlungssystem, das mit einem Datenbankmechanismus gekoppelt ist, zur Verfügung. Da über das ARPA-Netz viele der an der Sprachentwicklung beteiligten Stellen miteinander verbunden sind, hat es sich als eine große Hilfe bei der Vorbereitung von Dokumenten und Sitzungen oder bei der Durchführung von Auswertungen erwiesen.

Wenn die Arbeit am Terminal für einen Techniker zunächst auch sehr gewöhnungsbedürftig ist und die Aneignung einiger neuer Arbeitsgewohnheiten nötig macht, so erhöht sich doch die Effizienz von Teamarbeit durch die Verwendung eines solchen speichernden Kommunikationsmittels außerordentlich.

3. DIE SPRACHENTWICKLUNG DES US-DEPT. OF. DEF. IM VERGLEICH MIT PEARL

3.1 Zielsetzung

In diesem Abschnitt soll der Versuch eines Vergleiches zwischen PEARL und der zukünftigen gemeinsamen Programmiersprache des US-Verteidigungsbereiches gemacht werden. Dies liegt nahe, da beide Sprachen sich in ihrem Hauptanwendungsgebiet, der Programmierung von Realzeitsystemen, überlappen. Leider sind jedoch zur Zeit einem solchen Vergleich bezüglich seines Grades an Detailliertheit aus mehreren Gründen enge Grenzen gesetzt:

- Die Sprachentwürfe der Phase I sind zu großen Teilen als überholt zu betrachten, da während der Bewertungsphase Änderungsvorschläge und zum Teil berechtigte Kritik in so großem Umfang eingingen, daß die Sprachen vermutlich erhebliche Veränderungen erfahren werden, wenn auch nur ein kleiner Teil der Vorschläge berücksichtigt wird.

- Die technischen Anforderungen erfuhren beim Übergang von "revised IRONMAN" zum "STEELMAN"-Dokument erhebliche Änderungen. Manche davon, wie z.B. die das Tasking-Modell betreffenden, waren sogar grundsätzlicher Art.

- Über den derzeitigen Zustand der Sprachentwürfe wird aus Wettbewerbsgründen selbstverständlich Stillschweigen bewahrt.

Unter all diesen Einschränkungen kann aber doch versucht werden, einige generelle Unterschiede zu identifizieren.

Zunächst sind Entstehungsgeschichte und Zielsetzung der beiden Sprachen völlig unterschiedlich. PEARL wurde auf Initiative von Anwendern in enger Zusammenarbeit zwischen Herstellern, Softwarehäusern und Anwendern entwickelt. Ein Ziel dabei war, die Kommunikationslücke zwischen dem Spezialisten mit dem Wissen um den Prozeß und dem Datenverarbeitungsspezialisten dadurch zu schließen, daß dem Prozeßentwickler, sei es nun der Ingenieur, der Physiker oder der Chemiker, ein Mittel in die

Hand gegeben wird, das ihm erlaubt, einen großen Teil der anfallenden Programmieraufgaben selbst zu erledigen. Dazu war ein Instrumentarium nötig, da es gestattete, weitestgehend von speziellen Eigenschaften des Rechnersystems einschließlich der Interfacehardware zu abstrahieren. Dafür wurde an manchen Stellen eine gewisse Inflexibilität in Kauf genommen. Außerdem war es notwendig, dem "gelegentlich Programmierer" einen gewissen Komfort zu bieten, der natürlich manchmal mit entsprechendem Implementationsaufwand erkauft werden muß.

Die Sprachentwicklung des US-Dept. of Def. war dagegen von vornherein für den rein professionellen Programmierer gedacht. Sie soll speziell auch für die Erstellung von großen militärischen Realzeitsystemen eingesetzt werden können, bei denen Fragen der Programmzuverlässigkeit und der Verteilbarkeit des Arbeitsaufwandes eine große Rolle spielen. Außerdem war es wegen der großen Verschiedenartigkeit der Anwendungen nötig, besonders ausgefeilte Anpassungsmöglichkeiten an Charakteristika des Rechners und der Interfacehardware zu fordern. Auch spielen Effizienzfragen bei militärischen Anwendungen mit ihren manchmal drastischen physikalischen Einschränkungen eine größere Rolle als beim industriellen oder gar labormäßigen Rechnereinsatz.

3.2 Technische Gesichtspunkte

Es ist außerdem möglich, aus den im 'STEELMAN'-Dokument zusammengestellten Anforderungen einige technische Unterschiede zu PEARL abzuleiten, die die zukünftige Programmiersprache für den Bereich des US-Dod mit sehr großer Wahrscheinlichkeit aufweisen wird:

Zunächst wird sie 'kleiner' sein als Full-PEARL, da eines der Hauptprinzipien beim Entwurf ist, möglichst keinen Mechanismus in die Sprache einzubauen, der durch andere, bereits enthaltene Mechanismen dargestellt werden kann. Natürlich setzen Handlichkeit und Anwendbarkeit der Sprache der strengen Durchsetzung dieses Prinzips gewisse Grenzen.

Aus diesem Prinzip ergibt sich aber, daß die Sprache wohl kaum höhere Ein-/Ausgabeanweisungen wie in PEARL, ja selbst kaum solche wie in herkömmlichen Programmiersprachen enthalten wird. Vielmehr sollen benutzerorientierte Ein-/Ausgabefunktionen mit Hilfe der vorzusehenden Expansionsmechanismen aus einigen wenigen primitiven Operationen aufgebaut und dem Benutzer in (z.T. standardisierten) Anwenderbibliotheken zur Verfügung gestellt werden. Für den Benutzer soll dann aber ihr Aufruf nicht von dem in die Sprache eingebauter Funktionen unterscheidbar sein. Die allgemeine Form häufig gebrachter Ein-/Ausgabefunktionen soll aber schon in der Sprachbeschreibung festgelegt werden.

Auf leistungsfähige Mechanismen zur Definition von Typen, Operatoren, Modulen und Abstraktionen wird deshalb bei der Entwicklung der Sprache größtes Augenmerk verwandt werden müssen. Man darf wohl sagen, daß der Erfolg der Sprache in der Praxis mit der Qualität, Handlichkeit und Benutzerfreundlichkeit dieser Mechanismen stehen oder fallen wird.

Was die angebotenen Datentypen und darauf anwendbaren Operationen, das Prozedurkonzept und die Kontrollstrukturen angeht, so werden Unterschiede zu PEARL nur im Detail feststellbar und hauptsächlich durch das strenger eingehaltene Typkonzept bedingt sein.

Die im "STEELMAN" formulierten Anforderungen an die Konstruktionen zur Steuerung paralleler Prozesse würden von PEARL voll erfüllt werden, jedoch bleibt abzuwarten, wie diese Anforderungen von den Entwerfern ausgelegt werden, bevor irgendein Vergleich gezogen werden kann. Bei den Synchronisationsmechanismen ist – entsprechend dem Stand der wissenschaftlichen Diskussion auf diesem Gebiet – alles offen.

Die Mechanismen zur Behandlung von Fehlern und Ausnahmereaktionen werden in der "DoD-Sprache" voraussichtlich dem neuesten Stand der Technik entsprechen. Ein weiteres relativ neues Sprachmittel werden die "assertions" sein, die es gestatten, selbstkontrollierende Algorithmen zu schreiben. Ihre Semantik ist jedoch in PEARL durch entsprechende Verwendung des "SIGNAL"-Mechanismus nachbildbar.

Sprachmittel zur Beschreibung der statischen Systemkonfiguration außerhalb des eigentlichen Rechners, wie sie der Systemteil in PEARL zur Verfügung stellt, wird es nicht geben. Dafür wird es möglich sein, logische Datenstrukturen auf physikalische Speicherelemente im Rechner abzubilden. Ein Teil der Aufgaben des Systemteils, wie z.B. Bereitstellung von Steuerinformation für Betriebssystemgeneratoren oder Optimierungsparametern für Compiler, wird durch andere Sprachelemente übernommen. So ist z.B. an bedingte Übersetzung gedacht. Ein Grundprinzip, nämlich Trennung von maschinenabhängiger und maschinenunabhängiger Information, soll jedoch, ähnlich wie in PEARL, gelten.

4. LITERATUR

[1] B.W. Boehm et. al:
Information Processing / Data Automation Implication of Air Force Command and Control Requirements in the 1980s (CCIP-85), Vol. I, Highlights (Revised Edition), Febr. 1972
and:
Vol IV, Technology Trends: Software, Oct 1973, Space and Missile Systems Organization, AFSC, Los Angeles, California

[2] D.A. Fisher:
Automatic Data Processing Costs in the Defense Department;
Institute for Defense Analysis, Paper P-1046, AD-A004841, Oct. 1974

[3] Department of Defense Directive 5000.29, "Management of Computer Resources in Major Defense Systems", April 26, 1976

[4] (verschiedene Autoren)
Defense Management Journal, Vol. 11, No. 4, Oct 1975

[5] J. H. Williams, D. A. Fisher (editors):
Design and Implementation of Programming Languages – Proceedings of a DoD-Sponsored Workshop;
Lecture Notes in Comp.Sc., Vol. 54, Oct 1976, 496 pp
Springer-Verlag 1977

[6] Amoroso, Wegner, Morris, White:
Language Evaluation Coordinating Committee Report to the High Order Language Working Group (HOLWG);
Erhältlich auf Mikrofilm über die HOLWG oder von:
NTIS, US-Dept. of Commerce,
5285 Port Royal Road, Springfield, Va., 22161
AD-AØ37 634, Jan. 1977

[7] Department of Defense Instruction No 5000.31, "Interim List of DoD High Order Programming Languages (HOL)," Nov. 24, 1976

[8] P. Elzer:
Report on the 'review of analyses' phase I of the US-Dod-HOL-project;
LTPL-European group, paper PE 78032902

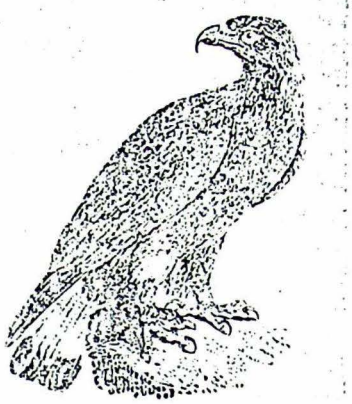
[9] DoD-HOL-Commonality Effort, Phase I Report; (4635 pages)
erhältlich auf Mikro-fiche über die 'HOLWG' oder von:
NTIS, US-Dept. of Commerce,
5285 Port Royal Road, Springfield, Va., 22161
ADB 95Ø 587

[10] Wulf, London, Shaw:
An Introduction to the Construction and Verification of ALPHARD Programs;
IEEE Transactions on Software Engineering, SE-2, 4, Dec. 1976
(pp. 253 - 265).

[11] Shaw, Wulf, London:
Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators; CACM, 2Ø, 8, Aug. 1977 (pp.553-564)

[12] D.E. Perry:
High level language features for handling I/O devices in real-time systems
PhD-Thesis, Stevens Institute of Technology, Castle Point, Hoboken,
New Jersey, Mai 1978

A N H A N G 1



DoD's common programming language effort is aimed at reducing the development and maintenance cost and improving the quality of software for embedded computer systems. Here is a brief review of the background, scope, goals, and methods of that effort.

DoD's Common Programming Language Effort

David A. Fisher
Institute for Defense Analyses

As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them—it has created the problem of using its products.

E. W. Dijkstra
Turing Award Lecture

As has often been noted, the past 25 years of digital computing have been characterized by striking increases in computing speed, memory capacity, and hardware reliability, with simultaneous decreases in power consumption and hardware cost. What is perhaps not so widely recognized is that these trends have led to inflated expectations for automating not only those tasks that had been previously performed manually, but also for automating some tasks that hadn't even been attempted before. Much of the burden of these increased expectations has fallen on software.

Within the Department of Defense, systems requirements for software have been expanded, as exemplified by automation of control functions in systems such as Tacfire, the Safeguard ballistic missile defense system, the Airborne Warning and Control System, the Trident ballistic missile system, and the Minuteman system.

Costs. Studies conducted in 1973 and 1974 provide some quantitative data on the size and makeup of the software problem.[1,2] Although little information is available, these studies give some conservative estimates that provide reliable lower bounds on the cost of software in the DoD. For example, in 1973 digital computer software costs were estimated at $3 billion to $3.5 billion annually and were growing in dollars and in proportion to other computer costs. An additional $2 to $3 billion were spent in the same year for the support and operation of computer systems. These studies also showed that the greatest software problems in the DoD, as measured by their cost, are associated with so-called embedded computer systems (Figure 1), and that the majority of costs are incurred in software maintenance rather than development.

The rising cost of computer resources has resulted in increased attention by the highest levels of management, and a number of technical and managerial procedures have been undertaken.[3] Initial guidance was provided by DoD Directive 5000.29, Management of Computer Resources in Major Defense Systems.[4]

At one time DoD was a major innovator and consumer of the most sophisticated computer hardware, but now it represents only a small fraction of the total market. In software, that unique position still remains: a significant fraction of the total software industry is devoted to DoD-related programs—and this is true in even larger proportion for the more advanced and demanding systems. Thus, as it once had for hardware technology, DoD now has the opportunity and responsibility to ensure that its influence on software technology is beneficial.

Common language effort. One of the major tasks undertaken by DoD to alleviate software problems

0018-9162/78/0300-0024$00.75

has been the common programming language effort. This effort is based on the idea that many of the support costs for software increase with the number of languages, and that languages must be suited to their applications. Furthermore, with a common programming language, a software development and maintenance environment could be built, providing centralized support and common libraries, that could be shared by several projects working in the same application area. Ideally, support software, including translators, could be developed in the source language so that any existing tools could be made available on a new machine at the cost of developing a new code generator for a standard compiler.

Embedded computer systems. Because the majority of software costs in the DoD are associated with embedded computer systems, the common language effort is concerned primarily with embedded computer software. The term "embedded computer system" was first used in 1974[5] to denote one that is logically incorporated in a larger system—e.g., an electromechanical device, a tactical system, a ship, an aircraft, or a communications system—whose primary function is not computation. Included in the concept of embedded computer systems is the support software necessary to design, develop, and maintain them. Computers used primarily for data processing, scientific, or research applications are not normally included in the embedded computer systems category.

Embedded computer software often exhibits characteristics that are strikingly different from those of other computer applications. The programs are frequently large (50,000 to 100,000 lines of code) and long-lived (10 to 15 years). Personnel turnover is rapid, typically two years. Outputs are not just data, but also control signals. Change is continuous because of evolving system requirements—annual revisions are often of the same magnitude as the original development.

Mission relationships. Software requirements vary from system to system depending upon the mission. The relative importance of execution efficiency, memory utilization, program modifiability, reliability, and program production time vary widely among applications and among components of a single system. Many embedded computer applications require software that will continue to operate in the presence of faults, whether the faults are in the computer hardware, input data, operator procedures, or the software.

At least 200 models of computers are used in embedded computer systems at DoD. In many applications, the computers must be installed in configurations that are incompatible with general-purpose installations. For example, the applications may require monitoring of sensors, control of equipment, display, or operator input processing. They must interface special peripheral equipment like radar, real-time clocks, and analog devices. Software must sometimes be able to respond at periodic (real time) intervals, to service interrupts within limited times, and to predict computation times. The time intervals vary from microseconds in device interface handling, through milliseconds in sensor monitoring, and seconds in control applications, to days in report generation.

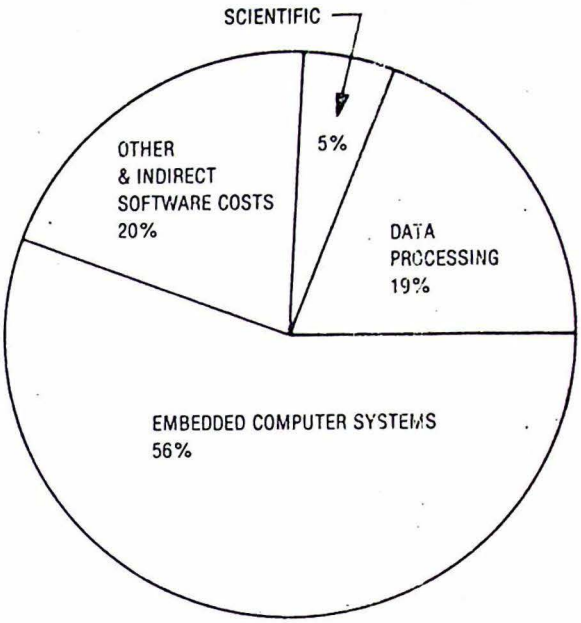


Figure 1. Breakdown of estimated $3 billion annual DoD software costs.

Special-purpose executive programs must be developed for many applications that cannot afford the overhead (and do not require the generality) of general-purpose operating systems. Systems programming capability is also needed to develop and maintain support software, including translators, software development tools, and testing aids, as well as their host operating systems.

In many applications, including command and control, training, and software development, it is necessary to access, manipulate, and display large quantities of data. Much of this data is symbolic or textual rather than numeric, and must be organized in an orderly and accessible fashion. Memory space rather than execution time is often the critical resource. On the other hand, a substantial numeric processing capability may still be essential, especially in simulation, sensor processing, and equipment control.

Software problems. Difficulties with embedded computer software are not atypical. Software problems that require or are susceptible to technical solution arise primarily from the nonsuitability of existing languages for embedded computer applications, from inadequate tools for software development and maintenance, and from insufficient concern for maintenance during software development.

Others, listed below, suggest management solutions in conjunction with technology.

Software development and maintenance are constrained by the availability of dollars, development time, machine resources, competent personnel, and useful programming tools. As with any activity in which expectations exceed the available capability, something must give. In this case, the symptoms appear in the form of software that is nonresponsive to user needs, unreliable, excessively expensive, untimely, inflexible, difficult to maintain, and not reusable.

Much has been said about the problems of software reliability. DoD software has all the common symptoms—occasional system crashes, inability to deal with user errors and ill-formed data, and errors which occur so frequently in a complex program from an apparently minor change. Software reliability, however, is particularly important in the military environment where errors can have severe consequences.

One little-recognized problem is that few useful software tools are available to the embedded software developer and maintainer. One reason is that resource limitations on hardware have led to an over-reliance on assembly language programming. There has been little incentive for individual projects to expend the effort and resources necessary to provide facilities that would be generally useful, especially when there are few, if any, other projects using the same programming language. This may also account for the lack of off-the-shelf software.

Finally, there is little cost accountability. This situation has been created by the lack of visibility of software to management, inaccessibility of software costs, and failure to give software the same scrutiny as hardware.

At least 450 general-purpose programming languages and (incompatible) dialects are used in DoD embedded computer applications—and none is widely used. With few exceptions the only common (i.e., widely-used) languages are Cobol (in data processing applications) and Fortran (in scientific and engineering applications). The remaining languages are used almost exclusively in embedded computer applications.

Programming languages

The present diversity of programming languages used in embedded computer systems did not cause most of the problems—nor would a common programming language cause them to disappear. Nevertheless, the existing language situation unquestionably aggravates them and inhibits some potential solutions.

The programming language is the central element in the design, development, and maintenance of software. It is the one software component that pervades all software component activity. It provides the building blocks from which software is constructed. Together with its implementation (as a compiler), it acts as the final arbitrator for the behavior of application software and associates an interpretation with each program. The programming language is a major concern when developing software tools and aids, when communicating techniques and algorithms, when writing manuals, and when training personnel.

Ill effects. The large number of programming languages and the lack of any widely-used language have had many ill-effects:

Excessive cost. There is enormous duplication of costs for the design, implementation, testing, maintenance, and training that must be repeated for the translators, software tools, application software, and support packages for each language.

Slow communication. Transfer of new software technology to practical use is severely retarded. The diversity of languages creates artificial boundaries that complicate communication, reduce understanding, and lead to mutual mistrust among users.

Scattered research. There is little research on the problems of software for embedded computer systems. Lack of programming language commonality makes it nearly impossible to gather quantitative data about problems that are unique to these applications.

Unnecessary ties to vendors. When a language is unique to a single project, so must be the support software. In consequence, the software maintenance is tied to the original vendor. This tendency is strengthened in the common situation in which the translator and support tools for the language are written in still another language that remains the property of the vendor.

Diversion from important tasks. The development of a new programming language for each project diverts energy from the real task. Of necessity, projects are concerned with their own application; their primary goal must be to develop the application software. Project personnel may have neither the inclination, time, funds, nor expertise to develop more powerful or more generally useful software tools that are needed to support their language.

Diffused expenditures. The large number of languages diffuses the available funds so that only the most primitive software aids can be afforded. Potentially useful software tools are limited to users of the associated language and, thus, provide little leverage.

Risk in using existing languages. When the existing languages are poorly supported (as they must be when there is no widely used language) and a new compiler must be developed for each new system (as is typical in embedded computer applications), the adoption of an existing language

by a new project is often more risky and less cost effective (at least during development) than it is to develop a new language specialized to the application.

History

The common language effort has had a short but lively history. It began in 1974 when groups in each of the military departments independently proposed the adoption of a common programming language for developing major defense systems. Those efforts included the Army "Implementation Language for Real-Time Systems" study, the Navy CS-4 effort, and the "High Order Language Standardization for the Air Force" study. In January 1975 a joint service program was formulated on the advice of the Director of Defense Research and Engineering.* He also instructed that no further funds be expended for the implementation of new programming languages in major defense systems until the problem of software commonality (i.e., of insufficient sharing of software resources) had been resolved.[6]

Working group. To coordinate the activities of the common language effort, a high order language working group was subsequently formed with official members from the Army, Navy, Air Force, Marine Corps, Defense Communications Agency, National Security Agency, and Defense Advanced Research Projects Agency. NASA and other offices within DoD have also participated. A representative of the British Ministry of Defence has been working full time in the United States since January 1977. The author acts as technical advisor. The high-order language working group is chaired by a representative of the Undersecretary of Defense, Research and Engineering.

The working group is chartered to "investigate the establishment of a minimal number of common high-order computer programming languages to be used in the development, acquisition, and support of computer resources embedded within Defense Systems."[7] In particular, it is to define the technical requirements for a common language, compare them with existing languages, recommend adoption or implementation of the necessary languages, and to monitor and assist any such actions. Thus, the working group coordinates all the activities of the common language effort but does not participate directly in the design or implementation of programming languages or their associated software.

The major concerns of the common language effort are to reduce the number of programming languages and to provide a useful, well supported environment for those that remain. The working group realized early that it would be impractical to convert existing programs to a common language; hence the common language effort applies only to new systems.

Interim list. A first step in reducing the number of programming languages was to adopt an interim list of approved languages. The military departments each nominated a limited number of languages. Those nominations resulted in the issuance of DoD Instruction 5000.31.[8] This instruction specifies that only approved high-order languages (Table 1) will be used to develop new defense system software, unless another language can be shown to be more cost effective over the system life cycle.

Table 1. Interim list of approved programming languages.

CMS-2
SPL-1
Tacpol
J3 Jovial
J73 Jovial
ANSI Cobol
ANSI Fortran

Generating the requirements. In the spring of 1975, the working group began the process of determining the characteristics of a general-purpose programming language suitable for embedded computer applications. The characteristics were to be given in the form of requirements which would act as constraints on the acceptability of a language, but would not dictate specific language features. The requirements are not a language specification; instead, they attempt to rigorously define the needed characteristics in a form that can be critically reviewed.

STRAWMAN. Although there are several widely accepted general goals and criteria (such as efficiency, reliability, readibility, simplicity, and implementability), they do not lend themselves to quantifiable assessment. At the opposite extreme are specific language features, advocated by some, which if adopted as requirements would impose strong constraints on the form but not necessarily increase the effectiveness of the language. The arguments for or against any specific language feature are often applicable to a class of features sharing certain properties, and they often depend on other characteristics of the language. The requirements attempt to isolate the needed properties from the features that implement them. Initially, rigorous definition at the level of requirements proved difficult, so a STRAWMAN of preliminary requirements was established. STRAWMAN was widely circulated within the military departments and to a lesser extent in the academic community and industry.

WOODENMAN. The reviews of STRAWMAN resulted in inputs which were formed into a fairly complete, but still tentative, set of requirements

*Now the Undersecretary of Defense, Research and Engineering, USDRE.

called WOODENMAN.[9] This document contained descriptions of the general (i.e., nonquantifiable) characteristics which were desired; it also contained many other desirable characteristics whose feasibility, practicality, and mutual compatibility had not been tested. WOODENMAN, too, was widely distributed, not only within the military departments but also to other government agencies, the computer science research community, and industry. Additionally, a number of technical experts outside the United States were solicited for comments, the European community being especially responsive.

TINMAN. Based on the various inputs and the official responses from each of the military departments, a TINMAN[10,11] set of requirements was derived. TINMAN removed former requirements for which there was no sound rationale, restricted unnecessarily general requirements, and modified others to be practical within existing technology. Each requirement in TINMAN had its own justification. TINMAN requirements were officially approved by the assistant secretary for research and development of each of the military departments in January 1976.

The document was circulated widely for comment, and in October of that year a workshop[12] was held at Cornell University to discuss the technical issues that had been raised by the requirements and to further investigate their feasibility.

IRONMAN. A new version of the requirements, called IRONMAN,[13] was issued in January 1977. IRONMAN requirements were substantially the same as those of TINMAN, but modified for feasibility and clarity and presented in an entirely different format. TINMAN was discursive and organized around general areas of discussion. IRONMAN, on the other hand, is very brief and is organized like a language description or manual. It is essentially a specification with which to initiate the design of a language. However, it is still sufficiently general to constrain the structure of a language without dictating the details of its design. A more recent revision, the Revised IRONMAN,[14] was issued in July 1977 and is available for comment.

At each iteration, comments were gathered and coordinated by the services and their working group representatives, then analyzed and reformulated as requirements by the Institute for Defense Analyses. In all, 74 commands and offices within DoD, 66 individuals outside of DoD, and 43 companies and organizations (not counting the workshop at Cornell or the language evaluation efforts) have contributed over 2000 pages of commentary on the requirements. Not all of the suggestions have been adopted, and many have been modified before acceptance, but each has been considered in sufficient detail to determine why it should or should not be followed.

Beginning with WOODENMAN, each iteration has reduced the number and generality of the capabilities requested. As the needs of the application have become better understood, as the application needs have been examined with respect to known language features, and as more emphasis has been placed on the general requirements for reliability, maintainability, and efficiency, many of the requirements have become both more precise and less restrictive.

Similarity of requirements. One surpising result of the requirements effort has been the similarity of the requirements among the different application areas. Early in this program, it appeared that different user communities might have fundamentally different requirements with insufficient overlap to justify a common language or might have critical requirements that were incompatible. Such communities include avionics, guidance, command and control, communications, and training simulators. However, it has been impossible to single out different sets of requirements for particular communities. Almost all the potential users had the same requirements, although priorities differed. Often the priorities varied among segments of a task. All users needed input-output, real-time facilities, strong data typing, etc.

Upon reflection, the technical rationale for this outcome was clear. The surprise was historical and was based on the observation that in the past the different communities have favored different language approaches. Further investigation showed that the origin of this disparity was primarily administrative rather than technical. This did not, however, establish that a single language could meet all the stated requirements, only that, if a language meeting all requirements were found, it would satisfy the perceived needs.

Language evaluation. During 1976, 23 programming languages (Table 2) were evaluated against the developing requirements. These evaluations were performed by 16 companies and organizations.

Most of the languages received at least two evaluations. In several cases the designers of a language were included among its evaluators. The report[15] consolidating the evaluations includes the following findings:

- No language satisfied the requirements so well that it could be adopted as a common language.
- Several of the languages were sufficiently compatible with the technical requirements so that they could be modified to produce an acceptable language. All of the languages in this group are derivatives of Algol-68, Pascal, or PL/I.
- Without exception, the evaluators found all the interim approved languages to be inappropriate as a basis for developing a common language.
- It was the consensus of the evaluators that it is currently possible to produce a single language that would meet essentially all the requirements.

Table 2. Examples of languages that were evaluated against the technical requirements.

1. Languages currently being used for embedded computer applications in DoD, such as
   - CMS-2
   - Jovial
   - SPL/1
   - Tacpol

2. Languages being used for process control and similar applications in Europe, such as
   - Coral 66
   - LIS
   - LTR
   - Pearl
   - RTL/2

3. Research languages known to satisfy specific requirements, such as
   - Euclid
   - Moral
   - ECL

4. Languages widely used outside DoD, such as
   - Cobol
   - Fortran
   - Pascal
   - PL/I

The latter finding means that no technological impediment to a single language was found and that it is likely that divergent requirements, such as those for readable programs, avoidance of unnecessary complexity, implementable compilers, semantic and syntactic consistency, machine independence, and object code efficiency, can be met.

As might be expected, the more modern languages tended to satisfy the requirements for reliability and maintainability, while languages intended for process control and DoD applications satisfied the requirements that reflect the special needs of embedded computer applications.

Design competition. Since no existing language simultaneously satisfied the needs of embedded computer applications, of reliable and maintainable software, and of machine independence, and since it appeared feasible to satisfy all the requirements without new technology, the services undertook a joint engineering design effort to produce a common language that would satisfy the requirements. Because all the languages that were identified as appropriate for modification are derivatives of Algol-68, Pascal, or PL/I, it was decided that the common language also should be a derivative of (but not necessarily upward compatible with) one of those three. Several competing designs were planned. Most of the fifteen proposals received, including the four best, were based on Pascal. These four—CII-Honeywell Bull, Intermetrics, SofTech, and SRI International—began parallel design efforts in August 1977.

We know that we design a language to simplify the expression of an unbounded number of algorithms created by an important class of problems. The design should be performed only when the algorithms for this class impose, or are likely to impose, after some cultivation, considerable traffic on computers as well as considerable composition time by programmers using existing languages. The language, then, must reduce the cost of a set of transactions to pay the cost of its design, maintenance, and improvement.

Alan J. Perlis
1966 Turing Award Lecture

The philosophy of the technical requirements

The technical requirements for the common language reflect six major goals: (1) that it be suitable for software in DoD embedded computer applications; (2) that it be appropriate for the design, development, and maintenance of reliable software for systems that are large, long-lived, and continually undergoing change; (3) that it be suitable as a common language (i.e., complete, unambiguous, and machine-independent standards can be established); (4) that it will not impose execution costs in applications where it provides unused or unneeded generality; (5) that it provide a base around which a useful software development, maintenance, and support environment can be built; and (6) that it be an example of good current language design practice. At the highest level, the requirements take the form of general design criteria (i.e., constraints) that are most strongly influenced by the first three goals.

Application needs. Many facilities must be provided in a language that is suitable for embedded computer applications, but four stand out because they are not usually provided in general-purpose languages for data processing and scientific applications:

User input-output interface specification. These applications use specialized input-output devices whose characteristics may not be known at the time of language design.

Exception handling. It must be possible to write programs that will automatically recover from errors, whether in the hardware, software, or data.

Real-time control. It must be possible to access real-time clocks, to control external devices in real time, and to respond within real-time constraints.

Parallel processing. It must be possible to write programs that control many devices in parallel, that share processors through interleaved execution, and whose parts may be executed concurrently on multiprocessors.

Needs of environment. The characteristics of military software and its environment impose several general design criteria on a suitable language:

Reliability. The combination of extremely complex systems with life-and-death implications may not be unique to the military, but it certainly requires that language characteristics which promote the production of reliable software be weighted very highly.

Modifiability. Perhaps as much as 90 percent of software costs in embedded computer systems go for software maintenance. Language features that contribute to the maintainability of reliable and efficient programs should have a major impact on software costs.

Efficiency. Physical limitations of military systems (e.g., an airplane) may impose limitations on the time and space for computations. In consequence, the efficiency of object programs is a legitimate and sometimes critical concern in military applications. Software that cannot meet these constraints may be, in effect, worthless.

Needs of commonality. Moreover, the desire for a language that can be widely used throughout DoD adds still more design criteria:

Machine independence. With over 200 computer models used in DoD, the language must be sufficiently machine-independent that it can be made available on a variety of object machines.

Practicality. The language must be sufficiently easy and inexpensive to implement that its wide use will be encouraged.

Complete definition. The language must have a complete and unambiguous definition to assure that software can be shared and incompatible implementations can be avoided.

Easily accessible support software. The availability of useful and easily accessible support software is, of course, the ultimate technical goal of the common language effort, but the ability to build such a support environment can be strongly influenced by the language characteristics.

General requirements. The design criteria were then translated into eight formal requirements dealing with the generality, reliability, maintainability, efficiency, simplicity, implementability, machine-independence, and formal definability of a suitable language. These eight, which constitute the first chapter of the technical requirements,[14] are further expanded into specific constraints on the design in the remaining chapters.

Attempts to expand the general requirements to a more detailed level where quantifiable measures could be applied, raised questions about the relative priorities of the general requirements and about how conflicts in requirements should be resolved. Some of the tradeoffs that were considered are outlined below. Others are given in references 11 and 16. Together they constitute a design philosophy for a common language, a philosophy of not making concessions on the general requirements unless absolutely necessary, and only after careful consideration of the implications.

Safety vs. efficiency. Intuition and historical observation tell us that there is a tradeoff between safety and efficiency in programs. Languages such as Euclid have emphasized safety but do not have efficient implementations. At the same time there are numerous examples in language designs of concessions to efficiency at the expense of safety (e.g., the "free union" in Pascal). The apparent tradeoff may not be inherent. Euclid was a vehicle for research and was not intended for use in large software efforts. The information needed to guarantee safety includes the type and ranges of values of variables (to limit their use in a program). That is, safety requires the same information as is needed by an optimzing compiler to determine what optimizations can be safely applied. This suggests that the same answer (i.e., languages that provide more information in programs) may provide partial solutions to the problems of safety, maintainability, and efficiency. This idea was pursued in the requirements development phase, and thus far no case has been found in which the efficiency of a correct program must be reduced in order to guarantee safety (although the compiler may be more complex).

Generalization vs. specialization. A general-purpose language can satisfy a variety of needs and can be applied to meet many, possibly unforeseen, situations, while a special-purpose language with built-in facilities for a particular application is often more efficient and therefore less expensive in use. The question is how to achieve both in the same programming language. The approach taken was to aim for a simple general-purpose language that would have the power needed for the intended applications, but would not yet be specialized for any particular application.

Such a language should have a few general-purpose structures, each providing a single primitive capability that can be combined with the others to form more specialized structures. Predefined application-oriented library definitions should be available in the language. As definitions made within the language, they can be independently controlled, need not add to the complexity of other applications, and need not affect the implementation of the language itself.

Programming ease vs. program safety. The more tolerant the programming language, the less it imposes on the programmer to specify his intent and assumptions in his programs, and thus the coding task is easier.

The safety of programs, on the other hand, is enhanced by requiring specification of the programmer's intent (e.g., specifying the range and types of variables), allowing redundant specifications (e.g., types determinable from either the formal or actual parameters), restricting the mixing of data types (e.g., prohibiting implicit type conversions), permitting restricted access to program components (e.g., specifying the scope of access for variables), and denying access to non-essential properties of data and programs (e.g., encapsulated type definitions).

A safe language allows the translator to check for program consistency and to verify that the programmer has, in fact, conformed to his stated intent and his own conventions in each program.

Considering that coding is a tiny fraction of the total software cost and that there are major software reliability and maintenance problems in embedded computer systems, the tradeoff between programming ease and program safety has been resolved in favor of safety.

Achieving efficiency. The desire for efficiency in software is often in conflict with other important goals such as minimal development cost, timely delivery, reliability, and functional utility. Systems requirements for efficiency ultimately take the form of space and time constraints imposed by the computer hardware. No additional benefit is derived from failure to use available space or failure to use an idle processor. Consequently, efficiency should be viewed as a constraint and not as an optimization criterion when developing programs.

Without automated software tools to identify which parts of a program are consuming the computational resources, the whole program must be optimized. Without efficient high-level languages suited to the task, the most capable programmers must be used to hand-tailor the machine code. The complexity of the task coupled with other constraints when developing a system seldom permits an optimal solution. More important, because a military system undergoes change throughout its lifetime, what may have initially been an efficient implementation becomes inefficient when changes occur in the assumptions and system characteristics against which it was optimized.

To be efficient, a high-order language must contain features that are appropriate to the applications. That is, it must have features that permit the user to express what is to be accomplished by the computation without dictating the details of how it is to be implemented. The translator can then select the most efficient implementation as a function of the generality and context of its use.

The language must be built from features that have efficient implementations on most machines; if the features are too general or too specialized, they often will not have efficient representations. It must be possible to combine built-in features to produce higher-level mechanisms that are specialized to a particular application, task, or program without imposing run-time cost for multiple levels of procedure calls. An efficient language will require the programmer to provide more formal documentation and will encourage the use of structured control primitives. Finally, whenever possible, features should be chosen to maximize the amount of processing that can be done during translation.

Current activities and plans

Three phases are planned for the design and implementation of the common language. The first phase or preliminary designs will be completed in February 1978. The preliminary designs will be incomplete but are supposed to be sufficiently detailed to determine the likelihood of their satisfying the major goals for the language. In particular, they are to address the most difficult design issues and to explain the rationale for each design decision.

The preliminary designs will be analyzed by a variety of teams from the military, industrial, and research communities on a voluntary basis between February 16 and March 13, 1978 (i.e., 390 individuals from 125 teams). The aim of the analyses[17] is to identify the major weaknesses, errors, and oversights in the preliminary designs and to determine their severity. The analyses will be used to identify the strengths and weaknesses of the individual designs, to obtain independent appraisals of the preliminary designs with respect to a number of specific design criteria, to help determine which subset of the design efforts will be continued into the second phase, and to provide feedback to the design contractors as they complete their designs. It is anticipated that the preliminary designs, the results of the individual analyses, and a summary evaluation report will be publicly available.

The candidate language designs will be completed in 1979, after which there will be an analysis and review of each design, leading to selection of one as the common language. A complete prototype translator (possibly in the form of an interpreter) will be available at that time.

The technical requirements continue to be refined with minor revisions issued at 6-month to 1-year intervals. The final version of the requirements and the final language design will be consistent with one another. Inconsistencies discovered during the preliminary design efforts are expected to impose changes in the requirements. A revised version, STEELMAN, is planned for late spring 1978.

During the remainder of 1978, the primary concern of the high-order language working group will be with the support and environment for a common language. Possible approaches will be identified and plans laid for language standards; translator certification; a root compiler; a common library; automated tools for software design, development, and maintenance; and a common (host) user-interface. A working paper outlining alternatives and initial positions will be issued by the working group in the

spring of 1978. Further input will be provided by the National Bureau of Standards technical symposium on "Tools for Improved Computing in the 80's" to be held in June 1978, and by a proposed workshop to be sponsored by the military departments following the NBS symposium.

Several parallel activities are planned for the third phase (i.e., the year following selection of the common language). These include test programming of DoD applications; fine tuning of the language design; and development of production compilers, the common library, support facilities, software development and maintenance tools, translator certification and test facilities, and special-purpose application libraries.

The language will not be made available for production use in DoD applications until the testing and implementation phase has been completed (i.e., 1980 in the current schedule). The common language, upon nomination by the military services, would then be added to the list of languages that are approved for use in DoD systems. No compiler will be certified until a standard definition of the language is adopted at the end of the third phase.

If the common language effort is successful (1) there will be a reduction in the number and a rise in the level of the general-purpose languages used for new software in DoD embedded computer systems, (2) there will be an effective and useful software development and maintenance environment built around the languages that remain, and (3) duplicate efforts to develop and maintain similar software tools and support systems will be reduced. The wider the acceptance and use of the language (inside and outside DoD), the greater will be the benefits to DoD. Its acceptance and usefulness, in turn, depend on its appropriateness for potential applications; on the quality of its design, implementation, and support; and on the economic implications of its use as seen by potential users. Consequently, the effort has encouraged and continues to encourage active participation from industry as well as from potential users within DoD. Interested organizations are encouraged to contribute to the continuing revision of the technical requirements, the development of a strategy to assure commonality among implementations of the language, and the planning and construction of a suitable environment for the language. ■

Acknowledgement

The work reported here was conducted under contract DAHC1573 C 0200 for the Department of Defense. The publication of this paper does not indicate endorsement by DoD, nor should the contents be construed as reflecting the official position of that agency.

Some of the material presented here has also appeared in reference 18.

References

1. Barry W. Boehm et al., *Information Processing/Data Automation Implication of Air Force Command and Control Requirements in the 1980s (CCIP-85).* Vol. I, *Highlights* (Revised Edition), February 1972, and Vol. IV, *Technology Trends: Software*, October 1973, Space and Missile Systems Organization, AFSC, Los Angeles, California.

2. D. A. Fisher, "Automatic Data Processing Costs in the Defense Department," Institute for Defense Analysis, Paper P-1046, AD-A004841, October 1974.

3. Barry C. DeRoze, "An Introspective Analysis of DoD Weapon System Software Management," *Defense Management Journal*, Vol. 11, No. 4, pp. 2-7, October 1975.

4. Department of Defense Directive 5000.29, "Management of Computer Resources in Major Defense Systems," April 26, 1976.

5. John H. Manley, "Embedded Computers—Software Cost Considerations," *AFIPS Conf. Proc., Vol. 41, 1974 NCC*, pp. 343-347.

6. Malcom R. Currie, "DoD Higher Order Programming Language," Memorandum issued by Director, Defense Research and Engineering (DDR&E), January 28, 1975.

7. "Charter for the High Order Language Working Group," Management Steering Committee for Embedded Computer Resources, Barry C. DeRoze, Chairman.

8. Department of Defense Instruction Number 5000.31, "Interim List of DoD High Order Programming Languages (HOL)," (signed) Frank A. Shrontz, Assistant Secretary of Defense (Installation and Logistics); Frank P. Wacher, Assistant Secretary of Defense (Comptroller); Richard M. Shriver, Director Telecommunication and Command and Control Systems; and Malcom R. Currie, Director of Defense Research and Engineering, November 24, 1976.

9. David A. Fisher, "*WOODENMAN*—Set of Criteria and Needed Characteristics for a Common DoD High Order Programming Language," Institute for Defense Analyses, Working Paper, August 13, 1975.

10. High Order Language Working Group, "Department of Defense Requirements for High Order Computer Programming Languages—'TINMAN'," June 1976.

11. D. A. Fisher, "A Common Programming Language for the Department of Defense—Background and Technical Requirements," Institute for Defense Analyses, Paper P-1191, AD-A028297, June 1976.

12. John H. Williams and David A. Fisher, Eds., *Lecture Notes in Computer Science, Vol. 54, Design and Implementation of Programming Languages—Proceedings of a DoD Sponsored Workshop*, October 1976, 496 pp., Springer-Verlag, 1977.

13. High Order Language Working Group, "Department of Defense Requirements for Higher Order Computer Programming Languages-IRONMAN," January 14, 1977.

14. High Order Language Working Group, "Department of Defense Requirement for High-Order Computer Programming Languages—Revised IRONMAN," July 1977.

15. S. Amoroso, P. Wegner, D. Morris, D. White, "Language Evaluation Coordinating Committee Report to the High-Order Language Working Group (HOLWG)," AD-A037634, 2617 pp., January 14, 1977, with appendices by

a. Lloyd Campbell, Army Ballistic Research Laboratory, Aberdeen, Maryland;

b. P. Parayre, Centre de Programmation de la Marine, Paris, France;

c. J. D. Ichbiah, CII-Honeywell Bull, Louveciennes, France;

d. Computer Sciences Corporation, Falls Church, Virginia;

e. A. Demers and J. Williams, Cornell University, Ithaca, New York;

f. Jean E. Sammet, Maurice Ackroyd, Michael L. Bell, I. Gray Kinnie, and Richard S. Kopp; IBM Federal Systems Division, Gaithersburg, Maryland;

g. Brian L. Marks, and Robert F. Maddock, IBM United Kingdom Laboratories, Winchester, England; and Tom C. Spillman, IBM Federal Systems Division;

h. J. G. P. Barnes, Imperial Chemical Industries Limited, Slough, England;

i. B. M. Brosgol, R. E. Hartman, J. R. Nestor, M. S. Roth, and L. M Weissman, Intermetrics, Inc., Cambridge, Massachusetts;

j. National Security Agency, Ft. Meade, Maryland;

k. Dr. Tomas Martin, PEARL Development Board, c/o Gesellschaft fur Kernforschung MBH, Karlsruhe, W. Germany;

l. RLG Associates, Inc., Reston, Virginia;

m. E. F. Miller and A. I. Wassermann, Science Applications, Inc., San Francisco, California;

n. John B. Goodenough, Clement L. McGowan, and John R. Kelly, SofTech, Inc., Waltham, Massachusetts;

o. Software Sciences Limited, Farnborough, Hampshire, England;

p. Texas Instruments Incorporated, Huntsville, Alabama.

16. David A. Fisher and Philip R. Wetherall "Rationale for Fixed-Point and Floating-Point Computational Requirements for A Common Programming Language," Institute for Defense Analyses, Paper P-1305, January 1978.

17. Defense Advanced Research Projects Agency, "Plan for the Analyses of the Preliminary Designs for A Common Programming Languages for the Department of Defense," December 30, 1977.

18. Lt. Col. William A. Whitaker, "The U. S. Department of Defense High Order Language Effort," Unpublished paper, 11 pp., Defense Advanced Research Projects Agency, November 9, 1977.

David A. Fisher is a member of the research staff at the Institute for Defense Analyses in Arlington, Virginia, where he has studied software costs in the DoD and provided technical assistance to the Common Programming Language Effort. He has been an assistant professor at Vanderbilt University and the University of Delaware. Previously he was a staff engineer at Burroughs Corporation where he was involved in the design of microprogrammable computers, of programming languages, of operating systems, and of military software.

His areas of publications include software costs, bounded-workspace algorithms, control structures, and parallel processing.

Fisher received a PhD in computer science from Carnegie-Mellon University in 1970, an MSE from Moore School of Electrical Engineering, University of Pennsylvania in 1967, and a BS in mathematics from Carnegie Institute of Technology in 1964. He is a senior member of IEEE and a member of AIAA and ACM.

Use order form on p. 105.

ANHANG 2

Decisions and Designs Incorporated, under contract to DARPA, produced an additional economic analysis of the DoD High Order Language strategy. This study was based upon the decision analysis techniques produced under previous DARPA contracts. The work used pieces of existing software and was implemented on the IBM 5100 desk top minicomputer. This work was briefed and demonstrated at the Management Steering Committee meeting of 20 October 1977. Models were actually rerun at the meeting break with parameter variations suggested by the committee. The program is available for further exercise on request. Three programs were involved:

- DECISION provides the decision tree, tracing the various alternative scenarios.

- SPREAD calculates the mixture of language usage as a result of those scenarios, growth of programs, distribution of projects by phases, etc.

- EVAL calculates the actual value of savings due to a particular language mix. It considers the difference between languages in a dozen different categories, quantitatively calculates their advantage over a baseline of Assembler, and sets up the commonality advantages as a function of the mix of language usage. The savings quoted are entirely software savings but there are also calculations of hardware savings implied by various language mixes.

All savings are compared against the baseline which would result from the exclusive use of Assembler; therefore, any model considering even the present use of high order languages, will exhibit savings. The impact of the common language program is therefore the difference between any proposed scenario with a common language, both savings and costs, less the savings calculated from present trends.

Technical savings expected even from the use of existing languages are quite significant. They are based upon some detailed consideration of the individual languages. Indeed, if significant savings were not expected, then the present policy towards high order languages would be ill-conceived. The proposed new common high order language is examined in detail and expected to give significantly higher technical savings. All savings are baselined against a flat software expenditure from now on of $3.2B per year and all savings are proportional to this number.

Runs were primarily directed towards examining the sensitivity of the introduction date and the rate of introduction. Detailed results are given in the attached figures. An introduction date of 1980 and total acceptance in 1985 may be compared with an introduction in 1985 and total acceptance in 1990 or with an introduction in 1980 and a slower acceptance, being complete only in 1990, etc. Both the saving rate per year and the integral savings to 1996 are shown. It must be noted that even the total acceptance of a language means that only new programs are being initiated and older programs in other languages continue for their life cycle; therefore, savings take some time to build up. On the other hand, the savings when established are enormous and pay back of language development costs can easily occur in much less than a year, once use is established.

An examination of the results based upon the total savings to 1996 gives the following trend:

- For a five-year introduction period, delay of the year of introduction through the period 1980 to 1987, with a corresponding delay of the complete adoption from 1985 to 1992, gives an average reduction in savings of about $1.5B per year delayed. This is simply the average magnitude of savings once full use is established.

- Keeping a constant total acceptance date of 1985, delay of the introduction from 1980 to 1982 costs about $1B per year delay.

- Having established an introduction date, slow acceptance or delay of the total usage date costs about $1B per year delay.

- All these figures are proportional to the total software costs envisioned. $1.5B corresponds to about 10% of the calculated additional savings due to the use of the new language. Thus a delay of one year in the introduction could only be justified if it resulted in an offsetting integrated savings of more than 10%, indicating an improved maximum savings rate increase of up to 20%.

It is recommended that the DoD Single Common Higher Order Language be introduced as rapidly as possible without penalizing technical quality or acceptability much more than 10%. Costs are small in all cases, being less than 1% of savings.

The High Order Language Working Group has established a third economic model for a DoD High Order Language commonality. This model is an outgrowth of the work done by Decisions and Designs Incorporated and its first version resembles the DDI model in many details. It has also received considerable input from discussions with the MITRE modelers. The main unique feature of the model is the fact that it is available on the ARPANET account of HOLWG for continued use and modification by the Working Group. It will continue to be maintained and updated during life of the program and will be used as a decision aid in the future. It is presently running on a PDP-10, is written in FORTRAN for transportability.

This model may be conceptually divided into three portions similar to the DDI structure but the elements are somewhat different.

- DECISION is a routine which allows the implicit input of events or decisions which can affect either the use of a particular language, the costs generated, or its effectiveness. Such decisions include the introduction of a language at a particular date, the rate of its adoption, the existence of control for that language, phasing out or restricting new starts in a language, permission to recode previous software from one language to another when cost effective, etc.

- PROS processes each one of these decisions and integrates an evolving world model up to the input time, including the amount of effort in each programming language by each Service, by size of program and phase of development and possibly application area. These are all derived numbers based upon input decisions and the resulting new starts. Ten languages are followed including all seven DODI 5000.31 languages plus Assembler, the common language and a lumped figure for all minor HOLs. For the baseline, a steady growth in the software produced from 1955 to date is postulated. This corresponds to past data. Future growth may continue or decrease as indicated by decision.

- EVAL gives an evaluation of the total value of the particular scenario at each time on the basis of benefits calculated from some thousand factors based on language phase of development and size of program. The coefficients were obtained informally from the organizations responsible for the individual languages. They are available for modification and it is expected that both the coeffieients and the definition of factors will evolve with continued study. The gross properties of the languages can be

checked against their general acceptance. For instance, a particular situation in which we find FORTRAN and JOVIAL to have similar utilization might be expected to correspond to similar magnitudes of technical benefit. Further, we can expect magnitude of technical benefit to be significant (factor of 2) in those cases where historically we see a large voluntary adoption rate. A 10% benefit would not be expected to have resulted in much acceptance, a 90% benefit should imply almost total adoption to the exclusion of Assembly language. All benefits (both positive and negative) and costs are reference to Assembly language.

Costs are calculated for training, compiler generation, introduction and maintenance, tools, and control. All costs are positive and, except for training, are not generated for Assembly language efforts. The benefits of commonality as calculated by the code are therefore evidenced in reduction in costs. These costs are then compared to technical gains to find the total benefit. Note that the benefit resulting from additional hardware commonality and from sharing and transportability of applications programs is not calculated in the present version but will be added later. These can be expected to give quite large savings but are also dependent on factors outside the pure software environment and will require a more inclusive world model.

The advantages of this model include its wide availability, its continued existence, the explicit use of decisions, a considerable accounting detail in the evolution of the world model, and the technical detail in the factors appropriate to the individual languages. It is expected that the model will continue to evolve and play a significant role in program decisions for this and other efforts.

The benefits calculated and the general results are only slightly less percentagewise than that of the DDI model and the resulting recommendations would be the same. It is improtant to note that all the technology commonality factors in the two programs were independently derived. This program, unlike the MITRE effort, is not deliberately conservative in the factors considered. It purports to be the current best estimate. It is, however, conservative in the omission of significant factors which have not yet been included.

The baseline case for this program normally considers an expanding software commitment, perhaps coupled with a decreasing cost of hardware, to increase the total fraction of computer resources devoted to software. Because of the benefits herein envisioned, the growth of functionality in software is calculated to be much more rapid than the growth of expenditures. With the inclusion of additional saving factors, it may be possible to get the cost of software in 1977 dollars to flatten out, even if the functionality increases by an order of magnitude.

ANHANG 3

DEPARTMENT OF DEFENSE

REQUIREMENTS FOR HIGH ORDER

COMPUTER PROGRAMMING LANGUAGES

"STEELMAN"

June 1978

PREFACE

The Department of Defense Common High Order Language program was established in 1975 with the goal of establishing a single high order computer programming language appropriate for DoD embedded computer systems. A High Order Language Working Group (HOLWG) was established to formulate the DoD requirements for high order languages, to evaluate existing languages against those requirements, and to implement the minimal set of languages required for DoD use. As an administrative initiative toward the eventual goal, DoD Directive 5000.29 provides that any new defense systems should be programmed in a DoD approved and centrally controlled high order language. DoD Instruction 5000.31 gives an interim list of approved languages: COBAL, FORTRAN, TACPOL, CMS-2, SPL/1, and JOVIAL J3 and J73. Economic analyses that were used to quantify the benefits from increased use of high order languages, also showed that the rapid introduction of a single modern language would increase the benefits considerably. The requirements have been widely distributed for comment throughout the military and civil communities, producing successively more refined versions from STRAWMAN through WOODENMAN, TINMAN, IRONMAN, and the present STEELMAN. During the requirement development process, it was determined that the single set of requirements generated was both necessary and sufficient for all major DoD applications. Formal evaluation was performed on dozens of existing languages concluding that no existing language could be adopted as a single common HOL for the DoD but that a single language meeting essentially all the requirements was both feasible and desirable. Four contractors were funded to produce competing prototype designs. After analysis of these preliminary designs the number of design teams was reduced to two. Their designs will be completed and a single language will emerge. Further steps in the program will include test and evaluation of the language, production of compilers and other tools for software development and maintenance, control of the language, and validation of compilers. Government-funded compilers and software tools, as well as the compiler validation facility, will be widely and inexpensively available and well maintained.

THE TECHNICAL REQUIREMENTS

The technical requirements for a common DoD high order programming language given here are a synthesis of the requirements submitted by the Military Departments. They specify a set of constraints on the design of languages that are appropriate for embedded computer applications (i.e., command and control, communications, avionics, shipboard, test equipment, software development and maintenance, and support applications). We would especially like to thank the phase one analysis teams, the language design teams, and the many other individuals and organizations that have commented on the Revised Ironman and have identified weaknesses and trouble spots in the technical requirements. A primary goal in this revision has been to reduce the complexity of the resulting language.

This revision incorporates the following changes. Care has been taken to ensure that the paragraph numbers remain the same as in the Revised Ironman. There have been several changes in terminology and many changes in wording to improve the understandability and preciseness of the requirements. Several requirements have been restated to remove constraints that were unintended but were implied because the requirement suggested a particular mechanism rather than giving the underlying requirement. The requirements for embedded comments (2I), unordered enumeration types (3-2B), associative operator specifications (7D), dynamic aliasing of array components (10B), and multiple representations of data (11B) have been deleted because they have been found unnecessary or are not adequately justified. The minimal source language character set has been reduced to 55 characters to make it compatible with the majority of existing input devices (2A). The do together model for parallel processing has been found inadequate for embedded computer applications and has been replaced by a requirement for parallel processes (section 9). The preliminary designs have demonstrated the need for additional requirements for explicit conversion between types (3B), subtype constraints (3D), renaming (3-5B), a language distinction between open and closed scopes (5G), and the ability, but preferably not special mechanisms, to pass data between parallel processes (9H), to write nonverifiable assertions (10F), to wait for several signals simultaneously (9J), and to mark shared variables (9C).

The Steelman is organized with an outline similar to that expected in a language defining document. Section 1 gives the general design criteria. These provide the major goals that influenced the selection of more specific requirements in later sections and provide a basis for language design decisions that are not otherwise addressed in this document. Sections 2 through 12 give more specific constraints on the language and its translators. The Steelman calls for the inclusion of features to satisfy specific needs in the design, implementation, and maintenance of military software, specifies both general and specific characteristics desired for the language, and calls for the exclusion of certain undesirable characteristics. Section 13 gives some of the intentions and expectations for development, control, and use of the language. The intended use and environment for the language has strongly influenced the requirements, and should influence the language design.

A precise and consistent use of terms has been attempted throughout the document. Many potentially ambiguous terms have been defined in the text. Care has been taken to distinguish between requirements, given as text, and comments, given as bracketed notes.

The following terms have been used throughout the text to indicate where and to what degree individual constraints apply:

shall	indicates a requirement placed on the language or translator
should	indicates a desired goal but one for which there is no objective test
shall attempt	indicates a desired goal but one that may not be achievable given the current state-of-the-art, or may be in conflict with other more important requirements
shall require	indicates a requirement placed on the user by the language and its translators (language is subject)
shall permit	indicates a requirement placed on the language to provide an option to the user (language is subject)
must	indicates a requirement placed on the user by the language and its translators (user is subject)
may	indicates a requirement placed on the language to provide an option to the user (user is subject)
will	indicates a consequence that is expected to follow or indicates an intention of the DoD; it does not in any case by itself constrain the design of the language
translation	refers to any processing applied to a program by the host or object machine before execution; it includes lexical analysis, syntactic error checking, program analyses, optimization, code generation, assembly, and loading
execution	refers to the processing by the object machine to carry out the actions prescribed by the program.

1. General Design Criteria

1A. Generality. The language shall provide generality only to the extent necessary to satisfy the needs of embedded computer applications. Such applications involve real time control, self diagnostics, input-output to nonstandard peripheral devices, parallel processing, numeric computation, and file processing.

1B. Reliability. The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors. The language shall require some redundant, but not duplicative, specifications in programs. Translators shall produce explanatory diagnostic and warning messages, but shall not attempt to correct programming errors.

1C. Maintainability. The language should promote ease of program maintenance. It should emphasize program readability (i.e., clarity, understandability, and modifiability of programs). The language should encourage user documentation of programs. It shall require explicit specification of programmer decisions and shall provide defaults only for instances where the default is stated in the language definition, is always meaningful, reflects the most frequent usage in programs, and may be explicitly overridden.

1D. Efficiency. The language design should aid the production of efficient object programs. Constructs that have unexpectedly expensive implementations should be easily recognizable by translators and by users. Features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for available generality where it is not needed, to maximize the number of safe optimizations available to translators, and to ensure that unused and constant portions of programs will not add to execution costs. Execution time support packages of the language shall not be included in object code unless they are called.

1E. Simplicity. The language should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. It should be as small as possible consistent with the needs of the intended applications. It should have few special cases and should be composed from features that are individually simple in their semantics. The language should have uniform syntactic conventions and should not provide several notations for the same concept. No arbitrary restriction should be imposed on a language feature.

1F. Implementability. The language shall be composed from features that are understood and can be implemented. The semantics of each feature should be sufficiently well specified and understandable that it will be possible to predict its interaction with other features. To the extent that it does not interfere with other requirements, the language shall facilitate the production of translators that are easy to implement and are efficient during translation. There shall be no language restrictions that are not enforceable by translators.

1G. Machine Independence. The design of the language should strive for machine independence. It shall not dictate the characteristics of object machines or operating systems except to the extent that such characteristics are implied by the semantics of control structures and built-in operations. It shall attempt to avoid features whose semantics depend on characteristics of the object machine or of the object machine operating system. Nevertheless, there shall be a facility for defining those portions of programs that are dependent on the object machine configuration and for conditionally compiling programs depending on the actual configuration.

1H. Complete Definition. The language shall be completely and unambiguously defined. To the extent that a formal definition assists in achieving the above goals (i.e., all of section 1), the language shall be formally defined.

2. General Syntax

2A. Character Set. The full set of character graphics that may be used in source programs shall be given in the language definition. Every source program shall also have a representation that uses only the following 55 character subset of the ASCII graphics:

```
 %&' ()*+,-./:;<=>?
0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ_
```

Each additional graphic (i.e., one in the full set but not in the 55 character set) may be replaced by a sequence of (one or more) characters from the 55 character set without altering the semantics of the program. The replacement sequence shall be specified in the language definition.

2B. Grammar. The language should have a simple, uniform, and easily parsed grammar and lexical structure. The language shall have free form syntax and should use familiar notations where such use does not conflict with other goals.

2C. Syntactic Extensions. The user shall not be able to modify the source language syntax. In particular the user shall not be able to introduce new precedence rules or to define new syntactic forms.

2D. Other Syntactic Issues. Multiple occurrences of a language defined symbol appearing in the same context shall not have essentially different meanings. Lexical units (i.e., identifiers, reserved words, single and multicharacter symbols, numeric and string literals, and comments) may not cross line boundaries of a source program. All key word forms that contain declarations or statements shall be bracketed (i.e., shall have a closing as well as an opening key word). Programs may not contain unmatched brackets of any kind.

2E. Mnemonic Identifiers. Mnemonically significant identifiers shall be allowed. There shall be a break character for use within identifiers. The language and its translators shall not permit identifiers or reserved words to be abbreviated. [Note that this does not preclude reserved words that are abbreviations of natural language words.]

2F. Reserved Words. The only reserved words shall be those that introduce special syntactic forms (such as control structures and declarations) or that are otherwise used as delimiters. Words that may be replaced by identifiers, shall not be reserved (e.g., names of functions, types, constants, and variables shall not be reserved). All reserved words shall be listed in the language definition.

2G. Numeric Literals. There shall be built-in decimal literals. There shall be no implicit truncation or rounding of integer and fixed point literals.

2H. String Literals. There shall be a built-in facility for fixed length string literals. String literals shall be interpreted as one-dimensional character arrays.

2I. Comments. The language shall permit comments that are introduced by a special (one or two character) symbol and terminated by the next line boundary of the source program.

3. Types

3A. Strong Typing. The language shall be strongly typed. The type of each variable, array and record component, expression, function, and parameter shall be determinable during translation.

3B. Type Conversions. The language shall distinguish the concepts of type (specifying data elements with common properties, including operations), subtype (i.e., a subset of the elements of a type, that is characterized by further constraints), and representations (i.e., implementation characteristics). There shall be no implicit conversions between types. Explicit conversion operations shall be automatically defined between types that are characterized by the same logical properties.

3C. Type Definitions. It shall be possible to define new data types in programs. A type may be defined as an enumeration, an array or record type, an indirect type, an existing type, or a subtype of an existing type. It shall be possible to process type definitions entirely during translation. An identifier may be associated with each type. No restriction shall be imposed on user defined types unless it is imposed on all types.

3D. Subtype Constraints. The constraints that characterize subtypes shall include range, precision, scale, index ranges, and user defined constraints. The value of a subtype constraint for a variable may be specified when the variable is declared. The language should encourage such specifications. [Note that such specifications can aid the clarity, efficiency, maintainability, and provability of programs.]

3.1. Numeric Types

3-1A. Numeric Values. The language shall provide distinct numeric types for exact and for approximate computation. Numeric operations and assignment that would cause the most significant digits of numeric values to be truncated (e.g., when overflow occurs) shall constitute an exception situation.

3-1B. Numeric Operations. There shall be built-in operations (i.e., functions) for conversion between the numeric types. There shall be operations for addition, subtraction, multiplication, division, negation, absolute value, and exponentiation to integer powers for each numeric type. There shall be built-in equality (i.e., equal and unequal) and ordering operations (i.e., less than, greater than, less than or equal, and greater than or equal) between elements of each numeric type. Numeric values shall be equal if and only if they have exactly the same abstract value.

3-1C. Numeric Variables. The range of each numeric variable must be specified in programs and shall be determined by the time of its allocation. Such specifications shall be interpreted as the minimum range to be implemented and as the maximum range needed by the application. Explicit conversion operations shall not be required between numeric ranges.

Approximate Arithmetic

3-1D. Precision. The precision (of the mantissa) of each expression result and variable in approximate computations must be specified in programs, and shall be determinable during translation. Precision specifications shall be required for each such variable. Such specifications shall be interpreted as the minimum accuracy (not significance) to be implemented. Approximate results shall be implicitly rounded to the implemented precision. Explicit conversions shall not be required between precisions.

3-1E. Approximate Arithmetic Implementation. Approximate arithmetic will be implemented using the actual precisions, radix, and exponent range available in the object machine. There shall be built-in operations to access the actual precision, radix, and exponent range of the implementation.

Exact Arithmetic

3-1F. Integer and Fixed Point Numbers. Integer and fixed point numbers shall be treated as exact numeric values. There shall be no implicit truncation or rounding in integer and fixed point computations.

3-1G. Fixed Point Scale. The scale or step size (i.e., the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable during translation. Scales shall not be restricted to powers of two.

3-1H. Integer and Fixed Point Operations. There shall be integer and fixed point operations for modulo and integer division and for conversion between values with different scales. All built-in and predefined operations for exact arithmetic shall apply between arbitrary scales. Additional operations between arbitrary scales shall be definable within programs.

3.2. Enumeration Types

3-2A. Enumeration Type Definitions. There shall be types that are definable in programs by enumeration of their elements. The elements of an enumeration type may be identifiers or character literals. Each variable of an enumeration type may be restricted to a contiguous subsequence of the enumeration.

3-2B. Operations on Enumeration Types. Equality, inequality, and the ordering operations shall be automatically defined between elements of each enumeration type. Sufficient additional operations shall be automatically defined so that the successor, predecessor, the position of any element, and the first and last element of the type may be computed.

3-2C. Boolean Type. There shall be a predefined type for Boolean values.

3-2D. Character Types. Character sets shall be definable as enumeration types. Character types may contain both printable and control characters. The ASCII character set shall be predefined.

3.3. Composite Types

3-3A. Composite Type Definitions. It shall be possible to define types that are Cartesian products of other types. Composite types shall include arrays (i.e., composite data with indexable components of homogeneous types) and records (i.e., composite data with labeled components of heterogeneous type).

3-3B. Component Specifications. For elements of composite types, the type of each component (i.e., field) must be explicitly specified in programs and determinable during translation. Components may be of any type (including array and record types). Range, precision, and scale specifications shall be required for each component of appropriate numeric type.

3-3C. Operations on Composite Types. A value accessing operation shall be automatically defined for each component of composite data elements. Assignment shall be automatically defined for components that have alterable values. A constructor operation (i.e., an operation that constructs an element of a type from its constituent parts) shall be automatically defined for each composite type. An assignable component may be used anywhere in a program that a variable of the component's type is permitted. There shall be no automatically defined equivalence operations between values of elements of a composite type.

3-3D. Array Specifications. Arrays that differ in number of dimensions or in component type shall be of different types. The range of subscript values for each dimension must be specified in programs and may be determinable at the time of array allocation. The range of each subscript value must be restricted to a contiguous sequence of integers or to a contiguous sequence from an enumeration type.

3-3E. Operations on Subarrays. There shall be built-in operations for value access, assignment, and catenation of contiguous sections of one-dimensional arrays of the same component type. The results of such access and catenation operations may be used as actual input parameter.

3-3F. Nonassignable Record Components. It shall be possible to declare constants and (unary) functions that may be thought of as record components and may be referenced using the same notation as for accessing record components. Assignment shall not be permitted to such components.

3-3G. Variants. It shall be possible to define types with alternative record structures (i.e., variants). The structure of each variant shall be determinable during translation.

3-3H. Tag Fields. Each variant must have a nonassignable tag field (i.e., a component that can be used to discriminate among the variants during execution). It shall not be possible to alter a tag field without replacing the entire variant.

3-3I. Indirect Types. It shall be possible to define types whose elements are indirectly accessed. Elements of such types may have components of their own type, may have substructure that can be altered during execution, and may be distinct while having identical component values. Such types shall be distinguishable from other composite types in their definitions. An element of an indirect type shall remain allocated as long as it can e referenced by the program. [Note that indirect types require pointers and sometimes heap storage in their implementation.]

3-3J. Operations on Indirect Types. Each execution of the constructor operation for an indirect type shall create a distinct element of the type. An operation that distinguishes between different elements, an operation that replaces all of the component values of an element without altering the element's identity, and an operation that produces a new element having the same component values as its argument, shall be automatically defined for each indirect type.

3.4. Sets

3-4A. Bit Strings (i.e., Set Types). It shall be possible to define types whose elements are one-dimensional Boolean arrays represented in maximally packed form (i.e, whose elements are sets).

3-4B. Bit String Operations. Set construction, membership (i.e., subscription), set equivalence and nonequivalence, and also complement, intersection, union, and symmetric difference (i.e., component-by-component negation, conjunction, inclusive disjunction, and exclusive disjunction respectively) operations shall be defined automatically for each set type.

3.5. Encapsulated Definitions

3-5A. Encapsulated Definitions. It shall be possible to encapsulate definitions. An encapsulation may contain declarations of anything (including the data elements and operations comprising a type) that is definable in programs. The language shall permit multiple explicit instantiations of an encapsulation.

3-5B. Effect of Encapsulation. An encapsulation may be used to inhibit external access to implementation properties of the definition. In particular, it shall be possible to prevent external reference to any declaration within the encapsulation including automatically defined operations such as type conversions and equality. Definitions that are made within an encapsulation and are externally accessable may be renamed before use outside the encapsulation.

3-5C. Own Variables. Variables declared within an encapsulation, but not within a function, procedure, or process of the encapsulation, shall remain allocated and retain their values throughout the scope in which the encapsulation is instantiated.

4. Expressions

4A. Form of Expressions. The parsing of correct expressions shall not depend on the types of their operands or on whether the types of the operands are built into the language.

4B. Type of Expressions. It shall be possible to specify the type of any expression explicitly. The use of such specifications shall be required only where the type of the expression cannot be uniquely determined during translation from the context of its use (as might be the case with a literal).

4C. Side Effects. The language shall attempt to minimize side effects in expressions, but shall not prohibit all side effects. A side effect shall not be allowed it it would alter the value of a variable that can be accessed at the point of the expression. Side effects shall be limited to own variables of encapsulations. The language shall permit side effects that are necessary to instrument functions and to do storage management within functions. The order of side effects within an expression shall not be guaranteed. [Note that the latter implies that any program that depends on the order of side effects is erroneous.]

4D. Allowed Usage. Expressions of a given type shall be allowed wherever both constants and variables of the type are allowed.

4E. Translation Time Expressions. Expressions that can be evaluated during translation shall be permitted wherever literals of the type are permitted. Translation time expressions that include only literals and the use of translation time facilities (see 11C) shall be evaluated during translation.

4F. Operator Precedence Levels. The precedence levels (i.e., binding strengths) of all (prefix and infix) operators shall be specified in the language definition, shall not be alterable by the user, shall be few in number, and shall not depend on the types of the operands.

4G. Effect of Parentheses. If present, explicit parentheses shall dictate the association of operands with operators. The language shall specify where explicit parentheses are required and shall attempt to minimize the psychological ambiguity in expressions. [Note that this might be accomplished by requiring explicit parentheses to resolve the operator-operand association whenever a nonassociative operator appears to the left of an operator of the same precedence at the least-binding precedence level of any subexpression.]

5. Constants, Variables, and Scopes

5A. Declarations of Constants. It shall be possible to declare constants of any type. Such constants shall include both those whose values are determined during translation and those whose value cannot be determined until allocation. Programs may not assign to constants.

5B. Declarations of Variables. Each variable must be declared explicitly. Variables may be of any type. The type of each variable must be specified as part of its declaration and must be determinable during translation. [Note, "variable" throughout this document refers not only to simple variables but also to composite variables and to components of arrays and records.]

5C. Scope of Declarations. Everything (including operators) declared in a program shall have a scope (i.e., a portion of the program in which it can be referenced). Scopes shall be determinable during translation. Scopes may be nested (i.e., lexically embedded). A declaration may be made in any scope. Anything other than a variable shall be accessable within any nested scope of its definition.

5D. Restrictions on Values. Procedures, functions, types, labels, exception situations, and statements shall not be assignable to variables, be computable as values of expressions, or be usable as nongeneric parameters to procedures or functions.

5E. Initial Values. There shall be no default initial values for variables.

5F. Operations on Variables. Assignment and an implicit value access operation shall be automatically defined for each variable.

5G. Scope of Variables. The language shall distinguish between open scopes (i.e., those that are automatically included in the scope of more globally declared variables) and closed scopes (i.e., those in which nonlocal variables must be explicitly imported). Bodies of functions, procedures, and processes shall be closed scopes. Bodies of classical control structures shall be open scopes.

6. Classical Control Structures

6A. Basic Control Facility. The (built-in) control mechanisms should be of minimal number and complexity. Each shall provide a single capability and shall have a distinguishing syntax. Nesting of control structures shall be allowed. There shall be no control definition facility. Local scopes shall be allowed within the bodies of control statements. Control structures shall have only one entry point and shall exit to a single point unless exited via an explicit transfer of control (where permitted, see 6G), or the raising of an exception (see 10C).

6B. Sequential Control. There shall be a control mechanism for sequencing statements. The language shall not impose arbitrary restrictions on programming style, such as the choice between statement terminators and statement separators, unless the restriction makes programming errors less likely.

6C. Conditional Control. There shall be conditional control structures that permit selection among alternative control paths. The selected path may depend on the value of a Boolean expression, on a computed choice among labeled alternatives, or on the true condition in a set of conditions. The language shall define the control action for all values of the discriminating condition that are not specified by the program. The user may supply a single control path to be used when no other path is selected. Only the selected branch shall be compiled when the discriminating condition is a translation time expression.

6D. Short Circuit Evaluation. There shall be infix control operations for short circuit conjunction and disjunction of the controlling Boolean expression in conditional and iterative control structures.

6E. Iterative Control. There shall be an iterative control structure. The iterative control may be exited (without reentry) at an unrestricted number of places. A succession of values from an enumeration type or the integers may be associated with successive iterations and the value for the current iteration accessed as a constant throughout the loop body.

6G. Explicit Control Transfer. There shall be a mechanism for control transfer (i.e., the go to). It shall not be possible to transfer out of closed scopes, into narrower scopes, or into control structures. It shall be possible to transfer out of classical control structures. There shall be no control transfer mechanisms in the form of switches, designational expressions, label variables, label parameters, or alter statements.

7. Functions and Procedures

7A. Function and Procedure Definitions. Functions (which return values to expressions) and procedures (which can be called as statements) shall be definable in programs. Functions or procedures that differ in the number or types of their parameters may be denoted by the same identifier or operator (i.e., overloading shall be permitted). [Note that redefinition, as opposed to overloading, of an existing function or procedure is often error prone.]

7B. Recursion. It shall be possible to call functions and procedures recursively.

7C. Scope Rules. A reference to an identifier that is not declared in the most local scope shall refer to a program element that is lexically global, rather than to one that is global through the dynamic calling structure.

Functions

7D. Function Declarations. The type of the result for each function must be specified in its declaration and shall be determinable during translation. The results of functions may be of any type. If a result is of a nonindirect array or record type then the number of its components must be determinable by the time of function call.

Parameters

7F. Formal Parameter Classes. There shall be three classes of formal data parameters: (a) input parameters, which act as constants that are initialized to the value of corresponding actual parameters at the time of call, (b) input-output parameters, which enable access and assignment to the corresponding actual parameters, either throughout execution or only upon call and prior to any exit, and (c) output parameters, whose values are transferred to the corresponding actual parameter only at the time of normal exit. In the latter two cases the corresponding actual parameter shall be determined at time of call and must be a variable or an assignable component of a composite type.

7G. Parameter Specifications. The type of each formal parameter must be explicitly specified in programs and shall be determinable during translation. Parameters may be of any type. The language shall not require user specification of subtype constraints for formal parameters. If such constraints are permitted they shall be interpreted as assertions and not as additional overloading. Corresponding formal and actual parameters must be of the same type.

7H. Formal Array Parameters. The number of dimensions for formal array parameters must be specified in programs and shall be determinable during translation. Determination of the subscript range for formal array parameters may be delayed until invocation and may vary from call to call. Subscript ranges shall be accessible within function and procedure bodies without being passed as explicit parameters.

7I. Restrictions to Prevent Aliasing. The language shall attempt to prevent aliasing (i.e., multiple access paths to the same variable or record component) that is not intended, but shall not prohibit all aliasing. Aliasing shall not be premitted between output parameters nor between an input-output parameter and a nonlocal variable. Unintended aliasing shall not be permitted between input-output parameters. A restriction limiting actual input-output parameters to variables that are nowhere referenced as nonlocals within a function or routine, is not prohibited. All aliasing of components of elements of an indirect type shall be considered intentional.

8. Input-Output, Formating and Configuration Control

8A. Low Level Input-Output. There shall be a few low level input-output operations that send and receive control information to and from physical channels and devices. The low level operations shall be chosen to insure that all user level input-output operations can be defined within the language.

8B. User Level Input-Output. The language shall specify (i.e., give calling format and general semantics) a recommended set of user level input-output operations. These shall include operations to create, delete, open, close, read, write, position, and interrogate both sequential and random access files and to alter the association between logical files and physical devices.

8C. Input Restrictions. User level input shall be restricted to data whose record representations are known to the translator (i.e., data that is created and written entirely within the program or data whose representation is explicitly specified in the program).

8D. Operating System Independence. The language shall not require the presence of an operating system. [Note that on many machines it will be necessary to provide run-time procedures to implement some features of the language.]

8E. Resource Control. There shall be a few low level operations to interrogate and control physical resources (e.g., memory or processors) that are managed (e.g., allocated or scheduled) by built-in features of the language.

8F. Formating. There shall be predefined operations to convert between the symbolic and internal representation of all types that have literal forms in the language (e.g., strings of digits to integers, or an enumeration element to its symbolic form). These conversion operations shall have the same semantics as those specified for literals in programs.

9. Parallel Processing

9A. Parallel Processing. It shall be possible to define parallel processes. Processes (i.e., activation instances of such a definition) may be initiated at any point within the scope of the definition. Each process (activation) must have a name. It shall not be possible to exit the scope of a process name unless the process is terminated (or uninitiated).

9B. Parallel Process Implementation. The parallel processing facility shall be designed to minimize execution time and space. Processes shall have consistent semantics whether implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor.

9C. Shared Variables and Mutual Exclusion. It shall be possible to mark variables that are shared among parallel processes. An unmarked variable that is assigned on one path and used on another shall cause a warning. It shall be possible efficiently to perform mutual exclusion in programs. The language shall not require any use of mutual exclusion.

9D. Scheduling. The semantics of the built-in scheduling algorithm shall be first-in-first-out within priorities. A process may alter its own priority. If the language provides a default priority for new processes it shall be the priority of its initiating process. The built-in scheduling algorithm shall not require that simultaneously executed processes on different processors have the same priority. [Note that this rule gives maximum scheduling control to the user without loss of efficiency. Note also that priority specification does not impose a specific execution order among parallel paths and thus does not provide a means for mutual exclusion.]

9E. Real Time. It shall be possible to access a real time clock. There shall be translation time constants to convert between the implementation units and the program units for real time. On any control path, it shall be possible to delay until at least a specified time before continuing execution. A process may have an accessible clock giving the cumulative processing time (i.e., CPU time) for that process.

9G. Asynchronous Termination. It shall be possible to terminate another process. The terminated process may designate the sequence of statements it will execute in response to the induced termination.

9H. Passing Data. It shall be possible to pass data between processes that do not share variables. It shall be possible to delay such data transfers until both the sending and receiving processes have requested the transfer.

9I. Signalling. It shall be possible to set a signal (without waiting), and to wait for a signal (without delay, if it is already set). Setting a signal, that is not already set, shall cause exactly one waiting path to continue.

9J. Waiting. It shall be possible to wait for, determine, and act upon the first completed of several wait operations (including those used for data passing, signalling, and real time).

10. Exception Handling

10A. Exception Handling Facility. There shall be an exception handling mechanism for responding to unplanned error situations detected in declarations and statements during execution. The exception situations shall include errors detected by hardware, software errors detected during execution, error situations in built-in operations, and user defined exceptions. Exception identifiers shall have a scope. Exceptions should add to the execution time of programs only if they are raised.

10B. Error Situations. The errors detectable during execution shall include exceeding the specified range of an array subscript, exceeding the specified range of a variable, exceeding the implemented range of a variable, attempting to access an uninitialized variable, attempting to access a field of a variant that is not present, requesting a resource (such as stack or heap storage) when an insufficient quantity remains, and failing to satisfy a program specified assertion. [Note that some are very expensive to detect unless aided by special hardware, and consequently their detection will often be suppressed (see 10G).]

10C. Raising Exceptions. There shall be an operation that raises an exception. Raising an exception shall cause transfer of control to the most local enclosing exception handler for that exception without completing execution of the current statement or declaration, but shall not of itself cause transfer out of a function, procedure, or process. Exceptions that are not handled within a function or procedure shall be raised again at the point of call in their callers. Exceptions that are not handled within a process shall terminate the process. Exceptions that can be raised by built-in operations shall be given in the language definition.

10D. Exception Handling. There shall be a control structure for discriminating among the exceptions that can occur in a specified statement sequence. The user may supply a single control path for all exceptions not otherwise mentioned in such a discrimination. It shall be possible to raise the exception that selected the current handler when exiting the handler.

10E. Order of Exceptions. The order in which exceptions in different parts of an expression are detected shall not be guaranteed by the language or by the translator.

10F. Assertions. It shall be possible to include assertions in programs. If an assertion is false when encountered during execution, it shall raise an exception. It shall also be possible to include assertions, such as the expected frequency for selection of a conditional path, that cannot be verified. [Note that assertions can be used to aid optimization and maintenance.]

10G. Suppressing Exceptions. It shall be possible during translation to suppress individually the execution time detection of exceptions within a given scope. The language shall not guarantee the integrity of the values produced when a suppressed exception occurs. [Note that suppression of an exception is not an assertion that the corresponding error will not occur.]

11. Representation and Other Translation Time Facilities

11A. Data Representation. The language shall permit but not require programs to specify a single physical representation for the elements of a type. These specifications shall be separate from the logical descriptions. Physical representation shall include object representation of enumeration elements, order of fields, width of fields, presence of "don't care" fields, positions of word boundaries, and object machine addresses. In particular, the facility shall be sufficient to specify the physical representation of any record whose format is determined by considerations that are entirely external to the program, translator, and language. The language and its translators shall not guarantee any particular choice for those aspects of physical representation that are unspecified by the program. It shall be possible to specify the association of physical resources (e.g., interrupts) to program elements (e.g., exceptions or signals).

11C. Translation Time Facilities. To aid conditional compilation, it shall be possible to interrogate properties that are known during translation including characteristics of the object configuration, of function and procedure calling environments, and of actual parameters. For example, it shall be possible to determine whether the caller has suppressed a given exception, the callers optimization criteria, whether an actual parameter is a translation time expression, the type of actual generic parameters, and the values of constraints characterizing the subtype of actual parameters.

11D. Object System Configuration. The object system configuration must be explicitly specified in each separately translated unit. Such specifications must include the object machine model, the operating system if present, peripheral equipment, and the device configuration, and may include special hardware options and memory size. The translator will use such specifications when generating object code. [Note that programs that depend on the specific characteristics of the object machine, may be made more portable by enclosing those portions in branches of conditionals on the object machine configuration.]

11E. Interface to Other Languages. There shall be a machine independent interface to other programming languages including assembly languages. Any program element that is referenced in both the source language program and foreign code must be identified in the interface. The source language of the foreign code must also be identified.

11F. Optimization. Programs may advise translators on the optimization criteria to be used in a scope. It shall be possible in programs to specify whether minimum translation costs or minimum execution costs are more important, and whether execution time or memory space is to be given preference. All such specifications shall be optional. Except for the amount of time and space required during execution, approximate values beyond the specified precision, the order in which exceptions are detected, and the occurrence of side effects within an expression, optimization shall not alter the semantics of correct programs, (e.g., the semantics of parameters will be unaffected by the choice between open and closed calls).

12. Translation and Library Facilities.

12A. Library. There shall be an easily accessible library of generic definitions and separately translated units. All predefined definitions shall be in the library. Library entries may include those used as input-output packages, common pools of shared declarations, application oriented software packages, encapsulations, and machine configuration specifications. The library shall be structured to allow entries to be associated with particular applications, projects, and users.

12B. Separately Translated Units. Separately translated units may be assembled into operational systems. It shall be possible for a separately translated unit to reference exported definitions of other units. All language imposed restrictions shall be enforced across such interfaces. Separate translation shall not change the semantics of a correct program.

12D. Generic Definitions. Functions, procedures, types, and encapsulations may have generic parameters. Generic parameters shall be instantiated during translation and shall be interpreted in the context of the instantiation. An actual generic parameter may be any defined identifier (including those for variables, functions, procedures, processes, and types) or the value of any expression.

13. Support for the Language

13A. Defining Documents. The language shall have a complete and unambiguous defining document. It should be possible to predict the possible actions of any syntactically correct program from the language definition. The language documentation shall include the syntax, semantics, and appropriate examples of each built-in and predefined feature. A recommended set of translation diagnostic and warning messages shall be included in the language definition.

13B. Standards. There will be a standard definition of the language. Procedures will be established for standards control and for certification that translators meet the standard.

13C. Completeness of Implementations. Translators shall implement the standard definition. Every translator shall be able to process any syntactically correct program. Every feature that is available to the user shall be defined in the standard, in an accessible library, or in the source program.

13D. Translator Diagnostics. Translators shall be responsible for reporting errors that are detectable during translation and for optimizing object code. Translators shall be responsible for the integrity of object code in affected translation units when any separately translated unit is modified, and shall ensure that shared definitions have compatible representations in all translation units. Translators shall do full syntax and type checking, shall check that all language imposed restrictions are met, and should provide warnings where constructs will be dangerous or unusually expensive in execution and shall attempt to detect exceptions during translation. If the translator determines that a call on a routine will not terminate normally, the exception shall be reported as a translation error at the point of call.

13E. Translator Characteristics. Translators for the language will be written in the language and will be able to produce code for a variety of object machines. The machine independent parts of translators should be separate from code generators. Although it is desirable, translators need not be able to execute on every object machine. The internal characteristics of the translator (i.e., the translation method) shall not be specified by the language definition or standards.

13F. Restrictions on Translators. Translators shall fail to translate otherwise correct programs only when the program requires more resources during translation than are available on the host machine or when the program calls for resources that are unavailable in the specified object system configuration. Neither the language nor its translators shall impose arbitrary restrictions on language features. For example, they shall not impose restrictions on the number of array dimensions, on the number of identifiers, on the length of identifiers, or on the number of nested parentheses levels.

13G. Software Tools and Application Packages. The language should be designed to work in conjunction with a variety of useful software tools and application support packages. These will be developed as early as possible and will include editors, interpreters, diagnostic aids, program analyzers, documentation aids, testing aids, software maintenance tools, optimizers, and application libraries. There will be a consistent user interface for these tools. Where practical software tools and aids will be written in the language. Support for the design, implementation, distribution, and maintenance of translators, software tools and aids, and application libraries will be provided independently of the individual projects that use them.

ANHANG 4

THE U.S. DEPARTMENT OF DEFENSE COMMON HIGH ORDER LANGUAGE EFFORT

William A. Whitaker, Lt.Col., USAF
Defense Advanced Research Projects Agency
1400 Wilson Blvd., Arlington, Va. 22209, USA

The United States Department of Defense (DoD) spends more than three billion dollars a year on computer software. This includes the design, development, acquisition, management, and operational support and maintenance of such software. Only a small fraction of this effort is involved with the accounting, inventory, payrolling, and financial management functions which are defined by the Federal Government as Automatic Data Processing, those functions that have their exact analogy in the commercial sector and share a common technology, both hardware and software. A much larger fraction of the DoD's computer investment is in computer resources which are embedded in, and procured as part of, major weapons systems, communications systems, command and control systems, etc. In this environment the DoD finds itself spending an even larger share of its systems resources on software. As a result, this area is receiving increasing attention from the highest levels of management. A number of technical and managerial initiatives have been called out to both reduce the cost and improve the quality of Defense systems software. A management plan has been formulated in this area and initial guidance is provided by DoD Directive 5000.29, Management of Computer Resources in Major Defense Systems.

In the area of software we may have, at the present time, more flexibility and a greater influence on the technology than with hardware. Some years ago, the DoD was a major innovator and consumer of the most sophisticated possible computer hardware. It now represents only a small fraction of the total commercial market. In software, that unique position still maintains. A significant fraction of the total software industry is devoted to DoD related programs and that is true in even larger proportion for the more advanced and demanding systems. Thus, there is both an opportunity and a responsibility in the software arena which is past for hardware.

One specific initiative which has been called out by DoD Directive 5000.29 is the use of high order languages (HOL) in systems development. The advantages are well known and in many communities, for instance, the COBOL financial management community or the FORTRAN scientific computational community, these advantages are so persuasive that there has been essentially no alternative to the use of these common languages for more than a decade. The obvious advantages include ease of writing of programs, self-documentation, ease of

maintenance, ease of modification, transportability of programs, simplification of training, etc.

It is surprising that a general consensus has not mandated a common high order language for embedded systems long since. There are, however, a number of managerial technical constraints that have acted against this in the past. For most Defense systems applications, very severe timing and memory considerations have been prominent in the past, often governed by real time interaction with the exterior environment. Because of these constraints, and restrictions in developmental cost and time scale, many systems have opted for assembly language programming. This decision is often substantially influenced by past experience with poor quality compilers and the fact that the assembler comes with the machine, while the compiler and its tools usually must be developed after the project has begun. The advantages of high order languages, however, are compelling and many more recent systems developments have turned to HOLs. Because of limitations of available high order languages, the programs generated most often include very large portions done in assembly code and linked to an HOL structure, negating many of the expected advantages.

Further, many systems have found it convenient to produce their own high order language or some perhaps incompatible dialect of an existing one. Since there is no general facility for control of existing languages, each systems office has had to do the configuration control on their language and compilers and continue to maintain such on their particular dialect through the entire maintenance phase of the system, which may be very long lived. This has had the effect of practically reducing the contractual flexibility of the government and restricting competition in maintenance and further development. This lack of commonality negates many advantages of high order languages including transportability, sharing of tools, the development of very powerful tools of high efficiency and, in fact, not only raises the total cost of existing tools, but in some cases essentially prices them out of the market. Many development projects are very poorly supported and forced to live with a technology which is far below the state-of-the-art.

By the early 1970's each of the military departments had underway studies or actual language designs which were expected to lead to common languages for large portions of those departments, in January 1975 the Director of Defense Research and Engineering set up a Defense-wide program with the goal of a single common military computer programming language for embedded systems. The intent was to have a real time language to supersede those numerous ones in existence while maintaining the standards of FORTRAN and COBOL, the success of which standards had provided impetus to this consolidation program. Further, to assure non-proliferation during the duration of this effort all other implementations of new high order programming languages for R&D programs were halted. A High Order Language Working Group (HOLWG) with representatives from DoD and the Military Services was established as the agent for this effort.

Briefly, the logic of this initiative is as follows:

- The use of a high order language reduces programming costs, increases the readability of programs, the ease of their modification,

facilitates maintenance, etc. and generally addresses many of the problems of life cycle program costs.

- A modern powerful high order language performs these tasks better and, in addition, may be designed to serve also in the specification phase and provide facilities for automatic tests and program verification. A modern language is required if real time, parallel processing, and input/output portions of the program are to be expressed in high order language rather than assembly language inserts which destroy most of the readability and transportability advantages of using an HOL. A modern language may also provide better error checking, more reliable programs, and the capability for more efficient compilers.

- Many of the advantages of a high order language can only be realized through computer tools. A total programming environment for the language includes not just compilers and debugging aids but text editors and interactive programming assistance, automatic testing facilities and proofs of correctness, extensive module libraries, and even semi-automatic programming from specifications. Universal use of those tools which are available today would significantly reduce the present cost of software. Development of more powerful tools holds even greater promise. Unfortunately, the average programmer's tool box is rather bare. Because of the difficulty of preparing these tools for each new language and machine and operating system, and the time involved, only the very largest projects have been able to assemble even a representative set. While in many cases development of tools can be shown to be desirable in the long run, day to day pressures usually prevail. There is almost never time to do it right. The use of a common high order language across many projects, controlled at some central facility, would allow the sharing of resources in order to make available the powerful tools which no single project could generate. It would even make those previously generated tools available at the beginning of a project, reducing start up time.

- Reducing the number of languages supported to a minimal number, therefore, provides the greatest economic benefit. There are, of course, costs associated with supporting any particular project and general costs of supporting the language. For a sufficiently large number of users, presumably the basic cost would be proportionally less. Perhaps 200 active projects contributing to a single support facility may not be proportionally much cheaper than two facilities each supporting 100 projects, although the absolute saving would be significant.

- There are, however, unique advantages to having a single military computer language. With a single language, one could reasonably expect new computers proposed for a project to be supplied by the manufacturer with a compiler. This is, in fact, the experience of the British with their common language effort. If there were five or

ten common languages, that is not a reasonable expectation. In fact, if there were a single common language, its use in DoD and the provision of tools by the DoD would make it a popular candidate for use elsewhere. Sufficient use could be generated that it would be economically sound to produce machines with firmware targeted to this high order language, decreasing cost and increasing efficiency. The multitude of military languages in the past has not received this sort of acceptance. A single powerful supported high order language might even be expected to influence academic curricula, improving the training not so much of individual programmers but the understanding and capabilities of the general engineering community for support of DoD programs.

The High Order Language Working Group (HOLWG) was chartered to formulate the requirements for common DoD high order languages, compare those requirements with existing languages, and recommend that adoption or implementation of the necessary common languages. In the very near term, administrative recourse has been taken. DoD Directive 5000.29 specifies that "DoD approved high order programming languages will be used to develop Defense systems software unless it is demonstrated that none of the approved HOLs are cost effective or technically practical over the system life cycle...Each DoD approved HOL will be assigned to a designated control agent..." Thus, the use of high order languages is established and indeed very strongly mandated, since life cycle costs are usually dominated by maintenance where the high order languages have considerable advantage over assembly language. Approved high order languages will be used, thereby reducing the proliferation and further, these languages will be controlled by central facilities. DoD Instruction 5000.31, Interim List of DoD Approved High Order Programming Languages, designates those languages and assigns control responsibility. COBOL and FORTRAN will be controlled by the Office of the Assistant Secretary of Defense (Comptroller) acting with The National Bureau of Standards and The American National Standards Institute. TACPOL shall be controlled by the Army. CMS-2 and SPL/1 shall be controlled by the Navy and JOVIAL J3 and J73 by the Air Force.

Formalization of these languages is a major step forward and recognizes for the first time the corporate commitment of the Department of Defense to provide support for languages in the long term. It stops the proliferation of languages in that all new systems are to be programmed in one of these languages, but there is no intent that already existing programs be redone or that the projects, already committed to a language, change. There are, however, limitations. The languages themselves are selected from the present Service inventories and are not, in general, modern powerful languages. They are generally deficient in tools and in availability of compilers. Further, we have only started on the concept of control. It will be some time before they reach the state of a well supported and controlled language. This is, therefore, an interim very near term solution. A more satisfactory technical solution to the problem is to formulate requirements, evaluate the existing languages, select the best for modification to meet the requirements, and build a single common high order language, if that proves technically feasible.

The first charge to the High Order Language Working Group was to establish

requirements. In terms of reference, this working group was to consider general purpose computer programming languages, those which are used by a programmer to talk to a computer, that is, of the level of the interim standards defined above. This is a limited goal and does not include either generalized requirements languages or very specific applications packages, which are formulated like languages but have only limited access to the capabilities of the computer. Such applications packages include simulation programs such as SIMSCRIPT, or GPSS, automatic test equipment languages such as ATLAS, which is really for communications between the test engineer and the technician, or special purpose packages for aerodynamics or civil engineering, or even generalized query languages or job control languages. Some of these are under study by other groups.

The goals of such a high order language are well agreed upon.

- One wishes to have the language facilitate the reduction of the cost of software. This cost must be reckoned on the total burden of the life cycle, including maintenance and certainly not just the cost of production or program writing.

- Transportability allows the reusing of major portions of software and tools from previous projects and the flexibility to modify hardware configurations.

- The maintenance of very long lived software in an ever changing threat situation requires responsiveness and timely flexibility.

- Reliability is an extremely severe requirement in many Defense systems and is often reflected in the high cost of extensive testing and verification procedures.

- The readability of programs produced for such long term systems use is clearly more important than coding speed.

- The general acceptability of high order languages is determined, at this time, by the efficiency and quality of the compiled code. While rapidly falling costs of hardware may make this difficult to substantiate in general, each project manager will compare the efficiency of the object code produced against an absolute standard of the best possible machine language programming. Very little degradation is acceptable.

While these and similar goals are well accepted, they do not lend themselves to a quantifiable or rational assessment of languages. Alternatively, one could establish criteria which were excessively explicit, determining the form but not necessarily the capability of the language. Rigorous definition of the exact level of requirement proved difficult. Therefore, a STRAWMAN of preliminary requirements was established to define this level by illustration. The STRAWMAN was forwarded to the Military Departments, other government agencies, the academic community and to industry. Additionally, a number of technical experts outside the U.S. were solicited for comments, the European

community being especially responsive, all the more vital since language research has been much more active there than in the U.S. over the last decade.

The review of the STRAWMAN resulted in inputs from which were put together a fairly complete, but still tentative, set of requirements called the WOODENMAN. This too was widely distributed for comment. Based on various inputs and the official responses from each of the Military Departments, a TINMAN set was derived which then represented the desired characteristics for a high order computer programming language for the DoD.

Early in this program, there was the feeling that different user communities might have fundamentally different requirements with insufficient overlap to justify a common language between them. Such communities include avionics, weapons guidance, command and control, communications, training simulators, etc. In addition to the embedded computer applications, even the scientific and the financial management communities were solicited for requirements for completeness sake. The surprising result was that the requirements so generated were identical. It was impossible to single out different sets of requirements for different communities. All users needed input/output, real time capability, strong data typing for compiler checking, modularity, etc. Upon reflection, the technical rationale for this was clear. The surprise was historical, based on the observation that in the past the different communities had favored different language approaches. Further investigation showed that the origin of this disparity was primarily administrative rather than technical, and the result that a single set of requirements would satisfy a broad set of users became less of a surprise. This did not, however, establish that a single language could meet all the stated requirements, only that, if a language meeting all the requirements existed, it would satisfy the users needs.

Very wide distribution of the TINMAN followed and for a year comments were received on this document. An international workshop was held at Cornell University in the fall of 1976 to illuminate the current state of the art of programming language design and implementation. In January, 1977, a new version was issued called the IRONMAN. This is essentially the same set of requirements as the TINMAN, modified slightly for feasibility and clarity, but it is presented in an entirely different format. The TINMAN was discursive and organized around general areas of discussion. The IRONMAN, on the other hand, is very brief and organized like a language description or manual. It is essentially a specification with which to initiate the design of a language. It is still sufficiently general so as not to constrain a particular structure of the language but just its capabilities. The IRONMAN was revised in July, 1977, mainly to clarify the intent, but also to correct the few errors and inconsistencies that had been identified.

The next phase of the work was the evaluation of existing languages. This was begun in a formal fashion in the summer of 1976, at which time the current requirements document was the TINMAN. Differences between the TINMAN and the IRONMAN are sufficiently minor so as not to affect the conclusions of this evaluation. The purposes of the evaluation were: to examine the existing

languages and determine if one or a combination could satisfy the requirements; to determine on the basis of evaluation of existing languages whether the requirements themselves were feasible and valid; to determine if it was possible within the state-of-the-art to have a single language satisfying all these requirements; and to recommend the procedure for arriving at the desired minimal set of languages.

The languages included in the evaluation were those nominated to the Interim Standard List, languages in wide acceptance elsewhere, and certain modern languages offering advanced capabilities. The main set of languages was evaluated very formally by contracts in which each language was evaluated by more than one contractor and each contractor had several languages to evaluate, thus giving a cross check on the results. In addition, a number of individuals submitted detailed evaluations of specific languages with which they had a unique familiarity. All these evaluations consisted of a comparison of the language against each individual point of the TINMAN. They were not mere existence checks but the languages were also examined for feasibility of modification should a particular point not be met and for features beyond the TINMAN requirements.

The following languages received formal evaluations: FORTRAN, COBOL, PL/1, HAL/S, TACPOL, CMS-2, CS-4, SPL/1, J3B, J73, ALGOL 60, ALGOL 68, CORAL 66, PASCAL, SIMULA 67, LIS, LTR, RTL/2, EUCLID, PDL2, PEARL, MORAL, EL-1. Besides those languages receiving formal evaluation, a number of other languages were examined for specific features or as examples of modifications of these languages and contributed data to the feasibility and flexibility of the various language approaches. In addition, some, such as APL, were immediately excluded as being inappropriate for Defense systems programming.

Such was the bulk of these studies that a government committee was put together to analyze and compare the evaluations and to make recommendations consistent with them. These conclusions and recommendations were adopted unanimously by the High Order Language Working Group as the basis for the next phase of the project. The conclusions may be briefly summarized as follows:

- Among all the languages considered, none was found that satisfies the requirements so well that it could be adopted as the common language.

- All evaluators felt that the development of a single language satisfying the requirements was a desirable goal.

- The consensus of the evaluators was that it would be possible to produce a language within the current state-of- the-art meeting essentially all the requirements.

- Almost all the evaluators felt that the process of designing a language to satisfy all the requirements should start from some carefully chosen base language.

- Without exception, the following languages were found by the evaluators to be inappropriate to serve as base languages for a

development of the common language: FORTRAN, COBOL, TACPOL, CMS-2, JOVIAL J73, JOVIAL J3B, SIMULA 67, ALGOL 60, and CORAL 66.

- o Proposals should be solicited from appropriate language designers for modification efforts using any of the languages, PASCAL, PL/1, or ALGOL 68 as a base language from which to start. These efforts should be directed toward the production of a language that satisfies the DoD set of language requirements for embedded computer applications.

- o At some appropriate time some choice should be made among these design efforts to determine which are most worthy of being continued to completion.

The definition of a base language, which evolved during this procedure, was one which was familiar to the community so that a number of contractors could use it as a starting point and provide an audit trail for evolution of the desired language which could be compared between contractors by government personnel. Many contractors would pass near some intermediate existing language, a modification of one of the bases in the direction of the requirements. For instance, PEARL or HAL/S could be considered modifications in the PL/1 family towards our desired real time language. This does not mean that those deemed inappropriate as a base language are not perfectly adequate for their present operational use. Indeed, the presence of COBOL and FORTRAN, to which we are committed to on a long term basis, belies that implication. Nevertheless, these languages would not be good starting points in that they have basic inconsistencies with the requirements or have been superseded by more appropriate starting points.

At this point we had determined, as well as can be done on the basis of paper studies without actual construction of a language, that a single language could be constructed to meet the requirements, further, that this could be done with elements which are mutually consistent and within the demonstrated state-of-the-art. The next step in the project was, therefore, to provide a preliminary definition of a language. Alternatively this might be considered an elaborate feasibility proof. Such definition was to be informal but fairly complete and consider the cost and nature of implementations.

This preliminary definition was done using the IRONMAN Revised as a specification and drawing upon the previous work done on evaluations. The procedure was multiple competitive contracts, with the best products to be selected for continuation to full rigorous definition and developmental implementation.

In August 1977, four contracts were awarded to produce competitive prototypes of the common high order language. These awards came as a result of a request for proposal and offers received from fourteen firms, both U.S. and foreign. The successful contractors were CII-Honeywell Bull, Intermetrics, Softech, and SRI-International.

While different approaches were offered, all four winning contractors proposed

to start from the computer language PASCAL as a base. This thereby restricts the products in form and makes it somewhat easier to compare the results. We were prepared to deal with three different base languages, so the outcome was coincidental. It should be noted that the requirements against which the language is being designed are not the same as those driving PASCAL and the result should not be expected to be a superset of Pascal. However, there will be some family resemblence and care is being taken not to modify surviving Pascal forms without substantial reason.

The products of Phase I , the preliminary designs , were received in February 1978. The considerable interest that this project has generated in the outside community made it possible to seek technical input for the evaluation of these designs from the industrial and academic communities worldwide. Eighty volunteer analysis teams were formed and produced extensive technical analysis of the designs. The period available was quite short, but the designs were only preliminary and the purpose of the analyses was to determine which should be continued to completion. On the basis of these analyses, CII-Honeywell Bull, and Intermetrics were selected to continue and resume work in April 1978.

As a result of both the designs and the analyses, the requirements were updated in June 1976 to a STEELMAN version. Since this may logically be the final set of requirements, some care was taken to clean it up and particularly to remove apparent misunderstandings and discrepancies which surfaced as the result of the actual design of the four languages. The exceptionally rigorous review of the languages by the analysis teams in the context of the requirements was a further exceptional test. It was the specific goal of this revision to assure that the level of the requirements was properly functional, neither too specific nor too general. Some portions of requirements have been deleted or modified as a result of these reviews and the parallel processing requirements have been generalized. The document remains a set of realistic requirements for large-scale systems in the present state-of-the-art. It does not describe an abstract, ideal language but is limited to one dealing in operational realities. Restrictions on character set reflect the distribution of input devices in all communities. The GO TO, remains although restricted, in order to ensure acceptability in those communities where it is still widely used.

The second phase of the design will include a language manual and a complete description, a formal definition of the language, and a test translator which will allow some execution. The test translator is only an aid in development and testing of the language and is not intended to be a production level compiler. A final selection between these two designs will take place in April 1979.

A selection will then be made of the single successful contender and that language will undergo elaborate test and evaluation of the language (as opposed to test of compilers) by rewriting a number of existing well defined programs or systems to demonstrate its applicability and advantages. A number of production compilers will be contracted for from different sources. A major thrust of this effort is to make the products non- proprietary and widely

available. Compilers will be tested against benchmarks and certified and when available the language can then be added to the list of approved languages in DoDI 5000.31.

Most recently we have seen the compleation of three different economic analyses. These were targeted to questions of expectation of savings to resulting from the successful compleation of this program. They further examined various introduction strategies and rates. Significant savings were demonstrated, and these were magnified by rapid introduction.

As the language becomes available, those other vital steps to programming environment will be provided including control, training, and tools. A particular technology to be fostered in this area is that of the generation of efficient compilers. A number of techniques including root compilers, compiler-compilers and compiler factories will be developed with the eventual goal of making available very inexpensive techniques for producing a compiler in this language for a new machine. Certification, testing and comparison of compilers by the control facility will promote competition and make the government a more knowledgeable buyer. While there is no intent to force any existing project to reprogram in a new language, there might be occasions in which this could be very profitable. Translation aids will be developed for converting existing language programs into the new language. It is hoped that this new language will be so powerfully supported that it will be the language of choice for future systems. It should be experimentally available by 1979 and available for general use in 1980.

These aspects of the development environment are being explored using procedures similar to that for the language requirements. A document has been prepared addressing those features of controlled support which would be required for the optimal utilization of the language. These may be requirements in a diffeent sense from the rigorous complete set of the STEELMAN, but the iteration methodology is also appropriate here. An initial version, SANDMAN, was superseded July 1978 by the PEBBLEMAN. This is still quite preliminary. Comment is being solicited from the software development community, a somewhat different group than the language experts who primarily address the language requirements. A meeting was held at the University of California (Irvine) in June 1978 discussing all aspects of the requirements. In September 1978, a meeting at Eglin AFB, Florida, discussed primarily the technology of retargetable compilers.

PEBBLEMAN proposes a configuration control board for the language whose responsibility is to maintain the definition and resolve any possible questions. A language control facility would provide validation and certification of compilers that they conform to the official definition within the limits of the current ability to test. The bulk of the document is concerned with defining those tools which could be provided in common to the use of language and outlining the methodology for producing and inter-relating those tools.

There is no intent to imply that all possible language requirements have been uncovered. New environments, new machines, new computer science and technology will eventually render the best language obsolete. It is rather surprising that everything required can be presently met. That may be the result of setting our sights on what we know. George Washington didn't ask for airplanes or atomic bombs or lasers, all he wanted was more muskets, cannon and sabers. Future language research is vital to the continued growth of capabilities. It is not the intent that the existence of this language stifle such research, rather that it provide a target and a user, a data and requirements gathering agency that will be able to survey the state-of-the-art and both direct and apply future developments.

Besides the normal interaction between portions of the Department of Defense and other agencies of the U.S. Government, this effort has had close relations with and received a great deal of support and technical input from a number of outside organizations with similar aims. The appropriate subcommittees of the American National Standards Institute and the International Standards Organization including their Working Group on Programming Languages for the Control of Industrial Processes have been kept closely informed of this work. The International Purdue Workshops on Industrial Computer Systems have long held an interest in this area and in particular an affiliate group, Long Term Procedural Language-Europe (LTPL-E) has as a goal the production of a language much like the one we desire. The goals of this group have recently been adopted by the European Economic Community and there has been a very intimate relationship between this group and the HOLWG. This is perhaps the most closely analogous group, trying to satisfy the requirements of several countries in many applications areas.

Perhaps the most successful national common language effort has been that of the British Ministry of Defense in specifying language CORAL 66 for all MOD real time applications. The HOLWG has received much valuable technical and managerial insight from the British experience and to enhance this cooperation, the British have assigned a senior technical expert to the HOLWG to be resident in Washington, providing both technical input and liaison. More recently, both the German and French governments have initiated procedures to standardize on existing high order languages, respectively, PEARL and LTR. The Federal Republic of Germany has also assigned a technical represenative to the HOLWG in Washington. The Japanese government, Ministry of Information, Technology and Industry, is subsidizing a consortium to produce a software production environment, central to which is a common programming language. The CCITT has proposed a common high order language for international use in communications.

It appears that the time is ripe for moving to a common high order language both technically and administratively, but significant milestones do remain. The High Order Language Working Group actively solicits comments and the cooperation the -rthein making this effort a success.

BIBLIOGRAPHY

Documents with AD number available from
the National Tecnical Infomation Service.

Department of Defense Directive 5000.29, Management of Computer Resources in Major Defense Systems,, 26 April 1976.

Department of Defense Directive 5000.31, Interim List of DoD High Order Programming Languages (HOL), 24 November 1976.

Department of Defense Requirements for High order Computer Programming Languages "TINMAN", June 1976.

Department of Defense Requirements for High order Computer Programming Languages "IRONMAN" Revised, July 1977.
Department of Defense Requirements for High order Computer Programming Languages "STEELMAN", June 1978

Institute for Defense Analyses, Automatic Data Processing Costs in the Defense Department, Paper p-1046, AD-A004841, D. A. Fisher, October 1974.

Institute for Defense Analyses, A Common Programming Language for the Department of Defense - Background and Technical Requirements, Paper P-1191, AD-A028297, D. A. Fisher, June 1976.

The Common Programming Language Effort of the Department of Defense, Paper given at the AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, 1 November 1977, D. A. Fisher.

Language Evaluation Coordinating Committee Report to the High Order Language Working Group (HOLWG), S. Amoroso, P. Wegner, D. Morris, D. White, AD-A037634, 14 January 1977.

Defense System Software Management Plan, AD-A022558, March 1976.

Defense System Software Research and Development Technical Plan, AD-A047062, September 1977.

The MITRE Corporation, A Cost/Benefit Analysis of High Order Language Standardization, M78-206, J.A. Clapp, E. Loebenstein, P. Rhymer, September 1977.

Decisions and Designs Incorporated, Benefit Model for High Order Language, TR78-2-72, Joseph M. Fox, March 1978.

High Order Language Working Group, Studies of the Economic Implications of Alternatives in the DoD High Order Commonality Effort, to be published.

HOLWG, DoD High Order Language Commonality Effort - Design, Phase I Report, ADB-950587, June 1978.

Lecture Notes in Computer Science, Vol. 54, Design and Implementation of Programming Languages - Proceedings of a DoD Sponsored Workshop, Ithaca, October 1976, Ed. John H. Williams and David A. Fisher, Springer-Verlag, 1977.

Report of the Irvine Workshop on the Environment for the DoD Common Language, NOSC, to be published.

Report of the Eglin Workshop on Common Compiler Technology, ADTC, to be published.

HOL PAST MILESTONES

A.	Formation of HOLWG	JAN 75
B.	STRAWMAN	APR 75
	WOODENMAN	AUG 75
C.	TINMAN	JAN 76
D.	DoD DIR 5000.29	APR 76
E.	DDR&E Directs Service Funding	MAY 76
F.	DoDI 5000.31	NOV 76
G.	Contractor Evaluation Completed	DEC 76
	Program Management Plan	JAN 77
H.	Report of Evaluation	JAN 77
I.	IRONMAN	JAN 77
	Final SOW for Design	MAR 77
	RFP Issued	APR 77
	Revised IRONMAN	JUL 77
	Design Contracts Awarded	JUL 77
J.	Economic Analysis	JAN 77 - NOV 77
K.	Language Design Phase I	AUG 77 - FEB 78
	Phase I Evaluation	FEB 78 - MAR 78
	Phase I Selection	APR 78
	Program Decision Date	APR 78
L.	STEELMAN	JUN 78
M.	Language Definition Phase II	APR 78 - MAR 79
	PEBBLEMAN Environment Req	JUN 78

HOL FUTURE MILESTONES

N.	Final Selection	APR 79
O.	Test & Evaluation	MAY 79 - DEC 79

P.	Language Refinement	MAY 79 - DEC 79
Q.	Development of Production Compilers	MAY 79---------->
	Compiler Maintenance	JAN 80---------->
R.	Control Facility/Tools	OCT 78 - OCT 79
	Control Facility All Up	FEB 80---------->
S.	Language Tool Development	MAY 79---------->
	Delivery of Tools	JUL 79---------->
	TOOLKIT Available	MAR 80
T.	Training Mat'l Definition	OCT 78 - MAY 79
	Training Courses	APR 79 - MAR 80----->
U.	Language Conversion Aids	JUL 79 - MAR 80
V.	Added to DoDI 5000.31	APR 80

ANHANG 5

DEPARTMENT OF DEFENSE

REQUIREMENTS FOR THE PROGRAMMING ENVIRONMENT FOR THE COMMON HIGH ORDER LANGUAGE

PEBBLEMAN

JULY 1978

PREFACE

The Department of Defense High Order Language Commonality language program was established in 1975 with the goal of establishing a single high order computer programming language appropriate for DoD embedded computer systems. A High Order Language Working Group (HOLWG) was established to formulate the DoD requirements for High Order Languages, to evaluate existing languages against those requirements, and to implement the minimal set of languages required for DoD use. As an administrative initiative toward the eventual goal, DoD Directive 5000.29 provides that new defense systems should be programmed in a DoD approved and centrally controlled high order language. DoD Instruction 5000.31 gave an interim list of approved languages including COBAL, FORTRAN, TACPOL, CMS-2, SPL/1, and JOVIAL J3 and J73. Economic analyses were used to quantify the benefits of going to high order languages and indicate considerably greater benefits associated with the rapid introduction of a single, modern language. The requirements have been widely distributed for comment throughout the military and civil communities, producing successively more refined versions from STRAWMAN through WOODENMAN, TINMAN, IRONMAN, and the present STEELMAN. During the requirement development process, it was determined that the single set of requirements generated was both necessary and sufficient for all major DoD applications. Formal evaluation was performed on dozens of existing languages concluding that no existing language could be adopted as a single common HOL for the DoD but that a single language meeting essentially all the requirements was both feasible and desirable. Four contractors were funded to produce competitive prototypes based upon PASCAL. A first-phase evaluation reduced the designs to two which will be carried to completion and from which a single language will emerge. Further steps in the program will be the test and evaluation of the language, production of compilers and a program development and tool environment, and control of the language and validation of compilers. The language validation facilities and government-funded compilers and tools will be widely and cheaply available to help promote use of the language.

TABLE OF CONTENTS

1	Introduction
1.1	Purpose
1.2	Reference Documents
1.3	Definition of Requirements Terms
2	Language Standard
2.1	Standard Document
2.2	Intent of the Common Language
2.3	Explicit Policy and Controls for Standardization
2.4	Approach
3	Control and Support Organizations
3.1	Configuration Control Board
3.2	Compiler Validation Facility
3.3	Language Support Facility
3.4	Application Libraries
3.5	User Organizations
4	Configuration Management
4.1	Objectives and Strategy
4.2	Configuration Management of the Common Language
4.3	Configuration Management of Compilers
4.3.1	General
4.3.2	Compile Validation Procedures
4.4	Configuration Management of Supporting Software
4.5	Configuration Management of Application Programs
5	Properties of Software tools
6	Design and Preparation Tools
6.1	General
6.2	Editors
6.3	Preprocessors
6.4	Design and Simulation
6.5	Automatic Translation Aids
7	Translation Tools
7.1	General properties
7.2	Technology
7.3	Interfaces
7.3.1	Handling of translators
7.3.2	Input to translators
7.3.2.1	Source statements
7.3.2.2	Control and option parameters

7.3.3 Output of translators
7.3.3.1 Code and control information
7.3.3.2 Listings
7.3.3.3 Error messages

8 Link/Load Tools
8.1 General requirements
8.2 Interfaces

9 Runtime Tools
9.1 General
9.2 Virtual language machine
9.3 Extended language machine
9.4 Runtime test and debug tools
9.4.1 Branch and timing counters
9.4.2 Trace and breakpoint
9.4.3 Interactive symbolic debugger
9.4.4 Symbolic dump

10 Maintenance
10.1 General
10.2 Maintenance oriented precautions
10.3 Individual tools

11 Management Tools
11.1 General
11.2 Libraries
11.3 Interface Monitor

12 Application Software
13 Training Support
13.1 Types of Training Required
13.1.1 Programmers Using the Common Language
13.1.2 Compiler Developers
13.1.3 Management of Projects Using the Common Language
13.2 Training Modes

14 Information Collection, Dissemination, and Promotion

Chapter 1

Introduction

1.1 Purpose

The Department of Defense (DoD) is defining a Common Higher Order Language (HOL) for embedded systems based upon a language requirements document. The Language Requirements Document was the product of the Military Departments coordinated by the DoD HOL Working Group (HOLWG). It incorporated comments and suggestions from the government, academic insti- tutions, and industry until judged to be of sufficient correctness and thoughness to be used as the requirements document for the design of the DoD Common High Order Language for embedded systems.

In order for the Common High Order Language to be successful in achieving the desired objectives, the environment in which it is used has to be conducive to its support. The environment includes all supporting activities and aids to develop programs for all systems applications - small, medium and large. These aids include for instance:

1. Organizations and methods to control the language and promote development of tools

2. Compilers for converting the HOL into the machine language of the target computer

3. Tools to aid in the design, test and debug of application programs

4. Organizations and methods to research the use of the language and prepare for follow on

5. Materials and techniques for training users of the language

6. Methods for collecting, cataloging and disseminating information about the language and programs written in the language

7. Project management aids to achieve successful implementation and maintenance

This document, titled PEBBLEMAN, describes the requirements for the environment necessary to the success of the Common High Order Language. It will go through a number of iterations, as the language requirements have, incorporating suggestions from all parts of the software community. It will also spin off more detailed requirements in specific areas such as software tools or control facilities.

The theme behind the inclusion of any topic has been to list all methods which

have come to be recognized as necessary for the production of reliable software.

This is a preliminary document for generating comments and wide latitude has been allowed in describing requirements. Later versions will strive for greater rigor.

Comments and further material are actively solicited from the reader. They may be transmitted directly to the HOLWG through its chairman:

Lieutenant Colonel William A. Whitaker
DARPA
1400 Wilson Boulevard
Arlington, Virginia 22209, USA

1.2 Reference Documents

o Standard Definition Document for the Common High Order Language (to be defined).

o DoD Requirements for High Order Computer Programming Languages, STEELMAN, June 1978.

o DoD Requirement for High Order Computer Programming Languages, IRONMAN, Revised July 1977.

o DoD Requirement for High Order Computer Programming Languages, TINMAN, June 1976.

o DoD High Order Language Program Management Plan, January 14, 1977.

o The Navy Fortran Validation System, Patrick M. Hoyt, AFIPS Volume 46, 1977.

o Design and Implementation of Programming Languages, DoD Sponsored Workshop, Ithaca 1976, Lecture notes in Computer Science Number 54, Springer - Verlag.

o DoD's Common Programming Language Effort, David A. Fisher, Computer, March 1978.

o Proceedings from Workshop on Environment and Control of DoD Common High Order Language, University of California, Irvine, June 1978, (to be published).

1.3 Definition of Requirements Terms

The following terms have been used throughout the text to indicate where and to what degree individual requirements apply.

Shall - indicates a requirement on the environment

Should - indicates a desired goal but one for which there is no objective test

May - indicates a requirement to provide an option to the user (user is subject)

Must - indicates a requirement placed on the user by the environment (user is subject)

Will - indicates a consequence that is expected to follow or indicates an intention of DoD

Chapter 2

Language Standard

2.1 Standard Definition Document

The syntax and semantics of the DoD Common High Order Language are to be described in a document which shall become the standard for deciding whether or not compilers conform to the language specification. That document shall be referred to as the Standard Definition Document. The Configuration Control Board shall maintain and interpret this document.

2.2 Critieria for the Language

The goal of the DoD Common High Order Language effort is to reduce total costs of software incurred by DoD. To this end a language is being designed with the following general criteria :

1. Generality - The language should be of a general nature applicable to a wide range of embedded systems computer applications.

2. Reliability - The language should promote, encourage, and enforce the use of techniques which lead to reliable software.

3. Maintainability - The language should emphasize readability and understandability of programs and lead to less costly maintenance.

4. Efficiency - The language should allow compilers which produce efficient object programs.

5. Simplicity - The language should reduce unnecessary complexity by means of uniform syntactic conventions and consistent semantic structure.

6. Implementation - The language should facilitate production of compilers that are easy to implement and are efficient.

7. Machine Independence - The language should strive for machine independence to make possible the trans- portability of application programs.

8. Formal Definition - There should be a formal definition of the language.

2.3 Explicit Policy and Controls for Standardization

In order for the HOL to achieve expected benefits, there shall be no variants of the Language. Organizations supporting the Common Language shall monitor and oppose any attempts at non-conformance to the published standard.

Once the Common Language is defined it will be added to the list of approved

higher order languages in DoD 5000.31.

A MIL-STD will be prepared and coordinated.

Registration of the Language as a Federal Information Processing Standard will be useful so that it may be used throughout the Federal Government.

The supporting organization will monitor activities in use of the Language and participate to promote further standardization of the Language. Submissions to the American National Standards Institute and the International Standards Organization may be appropriate to expand the user base and further reduce the likelihood of variants.

2.4 Approach

All environmental elements of this document support the above goals for the Common Language. Elements necessary for success are described in subsequent sections as follows.

1. The primary necessity is an organization to control the Language and promote development of its supporting software. Chapter 3 describes this organizational structure.

2. Methods for controlling the Common Language and its compilers are required to permit managed change when necessary for technical growth. These methods are described in Chapter 4.

3. Chapters 5 through 11 discuss the various types of tools which are necessary for the success of the Common Language.

4. Chapter 12 discusses requirements for application programs written in the Common Language. The methods described will lead to high portability for embedded systems.

5. Diverse training will be required for successful implementation of the Common Language. Chapter 13 addresses these training requirements.

6. Chapter 14 describes methods for collection and dissemination of Common Language materials (tools, compilers, training aids, etc.) so that the embedded computer software community will have ready access to all required Common Language information.

Chapter 3

Control and Support Organizations

The following paragraphs describe the organization of facilities and user groups which have been proposed to effectively control standardization of the DoD Common High Order Language and provide support.

3.1 Configuration Control Board

A Configuration Control Board (CCB) shall be established by DoD and be responsible for custody and maintenance of the standard definition of DoD Common High Order Language. Primarily, the function of the CCB shall be to minimize changes to the Language and prevent the occurence of variant translators.

The CCB shall be the final arbiter in any interpretation dispute of the Language definition. All official interpretations shall become part of the Language defination. All requests for changes and interpretations will receive a prompt response.

To reduce potential influence of special interest groups, the CCB shall be autonomous of compiler or applications developers.

Membership of the CCB may include representation from major Federal user communities within the United States. Expansion of the CCB to include representatives from outside the U.S. will be appropriate as other nations make a major commitment to the Language. Responsibility for the ccB may eventually be transfered outside the DoD.

The CCB shall be operational as soon as the language is frozen and submitted for standardization. At that time, formal definition of the DoD HOL shall be controlled by the CCB.

3.2 Compiler Validation Facility

A ficility shall be established to validate that compilers are complete and correct implementations of the DoD HOL. The facility shall not perform any function other than compiler validation and shall be independent reporting only to the CCB.

As a minimum, validation shall be conducted by subjecting compilers to a set of test programs. Development and maintenance of test programs shall be the responsibility of the validation facility. Trouble reports from users will be used to refine and update test programs in an effort to develop the most comprehensive test programs possible. All test programs shall be documented and made available to implementors who wish to test compilers independently prior to formal validation. Formal validation shall consist of reviewing compiler documentation and running the test programs. More elaborate validation will be conducted when such methodologies can be established.

A validation report shall be prepared and published by the facility.

Validation will be required by Defense projects when compilers are initially developed and when modified. A requirement will be to validate the compilers at the start of any project which plans to use the compiler.

The validation facility shall be established and operational within one year from the selection of the Common Language. When activity warrents, additional offices of the validation facility may be established.

3.3 Language Support Facility

A Language Support Facility (LSF) shall be established. It will be the focal point for most translator, support tool, and general library development and maintenance activity for the Common Language.

The LSF shall be the primary interface for user and implementor communities. IT will develop and maintain documentation, develop and conduct training courses and respond to all user and implementor inquiries.

All compilers, support tools, libraries, and language documentation maintained by the LSF shall be readily available to any legitimate and qualified language user or implementor.

3.4 Application Libraries

Application libraries, in the long term, will become very large and diverse. An effort will be undertaken to demonstrate the utility of a centrally supported application library. Several promising specialized application areas are signal processing, display processing, and communication networks. The ARPANET is a particularly well known application that may be provided as a common application package.

Once developed, application libraries could be supported and maintained by their specific support facilities. These facilities would be formed as desired for particular applications, in some cases colocated with the Language Support Facility.

3.5 User Organizations

User organizations are necessary to serve as a technical forum for common interests of those using the common language.

The Common Language Support Facility will foster user organizations by giving them recognition, disseminating information about their meetings and purposes, participating in their meetings, and giving due consideration to user organization proposals.

The organizations may be grouped by special interest such as the following:

1. Use of a particular compiler

2. Use of a particular computer

3. A particular application area

Chapter 4

Configuration Management

4.1 Objectives and Strategy

The strategy for managing the Common Language and Environment will be to control change, allowing for evolutional, stepped growth of the language, compiler, development tools, and test tools. Application programs will be controlled by the responsible agencies.

4.2 Configuration Management of the Common Language

Authorizations for all changes to the Common Language shall be under control of the Configuration Control Board (CCB). Except for errors, tThe CCB shall only implement changes which are upward compatible with the current Standard Definition Document.

All proposals for changes shall be accepted and recorded, but, in general, changes will not be undertaken due to the cost impact. Proposed changes will be investigated for being part of a generic requirement. The CCB will be supported by the LSF to investigate the impact and necessity for any proposed changes. Changes will be grouped and incorporated at the decision of the CCB. Either a time limit or a quantity of changes shall be used to make a change to the language.

4.3 Configuration Management of Compilers

4.3.1 General

All compilers used by the Federal Government, whether owned or not, must be controlled. Those owned by the Federal Government shall be controlled by a Common Language organization. Those not owned will be controlled through validation procedures which shall be required for all compilers used on federal projects. In both cases, complete descriptions of the product and accurate configuration information shall be maintained. Compilers will be validated after each modification. Compilers used outside the Federal Government are strongly encouraged to take advantage of validation facilities.

4.3.2 Compiler Validation Procedures

The purpose in validating the compilers is to ensure conformance to the Language as described in the Standard Definition Document. Conformance will be measured against the syntax and semantics of the language. The Common Language shall be specified in a manner which promotes validation and decreases the chances for misinterpretation by the developer.

The first method of validation shall be to compile and execute a standard series of programs written in the Common Language to test for correct

translation. The test set of programs will validate single statements and sequences. The tests will be comprehensive for all statements as well as for extremities and crucial cases. Tests will be made to assure that the limits of the Language Defination are not exceeded producing de facto extensions. The tests will also represent examples of embedded systems applications. A validation report shall be published stating the results of the tests and the resources used.

The testing shall be performed by the Language Validation Facility. The Validation Facility shall be responsible for preparing the set of standard tests, compiling the validation information, and approving the material. The Facility will either officially validate the translator or state necessary corrective actions prior to official validation. The implementor shall be required to certify that there are no unauthorized extensions to the language translator.

Bodies responsible for maintaining compilers should report new types of faults or bugs to the Validation Facility as the users are notified so that the test suite can be updated.

Any published test set must be recognized as being incomplete. Validation may be denied should the compiler fail any added tests.

As part of the validation effort, benchmark tests to describe compilation speed, compilation memory usage, object code speed, and object code memory usage will be required.

4.4 Configuration Management for Supporting Software

Support software that is owned by the LSF will be controlled by establishing procedures to identify the configuration and then managing approved changes. All such programs shall be recorded and cataloged by the LSF for promoting transferability.

4.5 Configuration Management of Application Programs

Application programs will be controlled by the development agencies. The development agencies will be requested to furnish the LSF with a description of appropiate new programs developed. The descriptions will be cataloged by the LSF. The description will be in a standard abstract form.

Development agencies may be required to search the catalog prior to developing new application programs in order to use existing, proven software rather than developing more.

Chapter 5

Properties of Software Tools

The Common Language should be designed to work in conjunction with a variety of useful software tools and application support packages. These will be developed as early as possible and will include editors, interpreters, diagnostic aids, program analyzers, documentation aids, testing aids, software maintenance tools, optimizers, and application libraries. There will be a consistent user interface for these tools. Where practical software tools and aids will be written in the language. Support for the design, implementation, distribution, and maintenance of translators, software tools and aids, and application libraries will be provided independently of the individual projects that use them.

The Common Language is only one tool in the whole design and development process for automated systems. A possible total structure of the development process may be described by the following steps:

A Understanding the Problem B Describing the Problem C Sketching a Solution D Refining the Solution E Identifying the Resources F Making the Solution Work F1 Development of Additional Hardware F2 Detailed Software Design F3 Coding F4 Component Test F5 Integration and System Test F6 Acceptance G Maintaining the Resulting System

Considerations in this document shall be restricted to tools which are related to the Common Language in a sense that they are either immediately necessary for its use, such as translators, help to improve its performance significantly, such as symbolic debuggers, or are uniquely related to the technology of the Common Language.

One of the most important goals is that tools should be machine-independent to the greatest possible extent and that they should run on the standard military computers so as to provide maximum useability.

There will be a modular set of tools which can be tailored to the individual needs of applications and which can be updated and extended according to technological progress. To ensure this the following measures will be necessary [Note that the order of the following points does not imply a sequence in time]:

- A common framework shall be established into which the software tools, libraries, and data base fit. This framework shall comprise the definition of a consistent methodology as well as the identification and eventual standardization of interfaces between tools.

- The tools, while modular with respect to the common framework, shall in themselves be developed in small pieces which can be easily composed to perform traditional software development and maintenance tasks, which should be able to

run individually on rather small machines, and which can be easily replaced by more advanced versions.

- The methodology shall facilitate iteration between development steps.

- Documentation shall be an integral part of all development steps. The tools shall be designed in a way that the methods used are either self-documenting or produce supporting documentation automatically, thus ensuring a proper flow of information between the design levels.

- Design and implemention of the tool environment should emphasize the problems of software maintenance. Further research should help to solve the urgent problems in this area.

- Semiautomated tools shall be utilized when fully automated tools are infeasible or too expensive.

- It should be possible to apply appropriate analysis and verification methods to each development step.

- Test and debug tools both for off-line and on-line (static and dynamic) testing shall be integrated into the program development system. They shall include methods for systematic testing which derive their control information from earlier development stages.

- The interfaces between tools (and between parts of tools) as well as the handling of the tools should be standardized for military applications.

- A list of essential tools which may be considered to make up a basic programmers workbench shall be provided with a description of what each will contribute to and expect from the data base.

- The production of new powerful tools for the Common Language by Industry and the academic community will be encouraged. Government supplied tools will be chiefly limited to those simple tools in most common use.

Chapter 6

Design and Preparation Tools

6.1 General

The proper analysis of the requirements for a problem solution, is highly dependent on the technology concerned. It can therefore not be expected that there is one uniform method by which the requirements for a system can be identified, formulated and specified. On the other hand, there should be methods and tools which guide, support and, if necessary, force the designer to express his requirements in a way which is unambiguous (at least as far as feasible), appropriate for further automated processing, where practical, and oriented towards computerized systems.

The development of a variety of tools should be encouraged, including textbooks, development standards, cookbooks, case studies, standardized representations, computer aided design systems, both off-line and interactive.

The methods and tools available in this area should also facilitate integrated design of hardware and software and should support structured decomposition. They should include graphic methods and representations and be fit for automation, preferrably in an interactive way. It might be envisaged that they support the more detailed levels of system design with a higher degree of automation, utilizing data bases which contain information on available system components and their method of interconnection. Simulation and testing techniques should be integrated in such a way that the design support system generates input information for simulation packages and the testing process.

6.2 Editors

Editor programs to allow generation and changes to source programs shall be required. These may be either batch or interactive. They must allow integration with the library system which maintains the files which are edited.

Editors shall optionally output source listings in a well structured standard format. The should allow local formatting standards and conventions to be incorporated. More elaborate versions may be able to produce flowcharts of a given program or to do some parsing in order to facilitate interactive program production by early recognition of certain errors.

Editors should optionally be able to produce compressed versions of source programs , e.g. in order to save file space .

6.3 Preprocessors

There may also be preprocessors for source programs, which, prior the translation phase, are able to test for compliance to project coding

standards, to capture data for tracing requirements throughout the project, and to symbolically execute the code for analysis.

They should also be able to optimize programs on source level for input to higher level language machines . There may also be preprocessors to transform higher level structured mechanisms or complete very high level application oriented languages into the Common Language .

6.4 Design and Simulation

Design tools which are related to the language should include application libraries which facilitate the composition of large systems from partially reuseable modules, interface description aids, stub-generators, tools for proof of correctness, etc.

The input information for simulation packages should be produced automatically from an analysis of the structure of the programs. Programs to generate simulation test data by analyzing the code shall be required. These programs will help to ensure that the test data provides full coverage (execution of all instructions) with the fewest possible test cases.

6.5 Automatic Translation Aids

While it will not be possible to take full advantage of the Language capabilities when transcribing from programs designed and implemented in other languages, there are occasions on which it will be desirable to translate some portions of an existing program from an old language into the new Common Language. Packages and techniques will be developed to facilitate this. In most cases complete automation is not required, just machine aid to a programmer. Languages on the DoD Instruction 5000.31 list are the best candidates for such aids.

Chapter 7

Translation tools

7.1 General Properties

Translators shall implement the standard definition. Every translator shall be able to process any syntactically correct program. Every feature that is available to the user shall be defined in the standard, in an accessible library, or in the source program.

Translators shall be responsible for reporting errors that are detectable during translation and for optimizing object code. Translators shall be responsible for the integrity of object code in affected translation units when any separately translated unit is modified, and shall ensure that shared definitions have compatible representations in all translation units. Translators shall do full syntax and type checking, shall check that all language imposed restrictions are met, and should provide warnings where constructs will be dangerous or unusually expensive in execution and shall attempt to detect exceptions during translation. If the translator determines that a call on a routine will not terminate normally, the exception shall be reported as a translation error at the point of call.

Translators shall fail to translate otherwise correct programs only when the program requires more resources during translation than are available on the host machine or when the program calls for resources that are unavailable in the specified object system configuration. Neither the language nor its translators shall impose arbitrary restrictions on language features. For example, they shall not impose restrictions on the number of array dimensions, on the number of identifiers, on the length of identifiers, or on the number of nested parentheses levels.

7.2 Technology of translators

Translators for the language will be written in the language and will be able to produce code for a variety of object machines. The machine independent parts of translators should be separate from code generators. Although it is desirable, translators need not be able to execute on every object machine. The internal characteristics of the translator (i.e., the translation method) shall not be specified by the language definition or standards.

In this area the best of existing and emerging technology will be used. Compilers shall be portable to the greatest possible extent, while separate code-generators shall allow inexpensive adaptation to various target machines. Host compilation is preferred because of its greater possibilities for optimization and its faster throughput. Self-hosted compilers should be developed where technical and organizatorial reasons prevent other solutions.

Interpreters and incremental compilers shall be provided for environments where

fast response times during testing are required. In case of errors the translators shall output the maximum possible amount of information compatible with their respective designs. Whereas it can not necessarily be expected that all translators have the same standardized error messages, attempts shall be made to standardize format and contents of such messages if they occur.

Input for compilers shall be prepared by editors and program structuring tools in such a way that the card-image is no longer a restriction for the formatting of source programs.

Translators shall optionally provide outputs which contain the information necessary to interface with runtime test and debug tools.

Translators shall be unforgiving in identifying all syntax and semantic errors.

Compilers shall generate efficient code.

Translators shall be validated as error free as far as practical given the state of technology.

Each translator shall have optimization features which may be used to optimize memory useage or execution speed.

Translators shall be written in a modular fashion which allows inclusion of approved language changes.

It is not intended generally to prescribe the type of intermediate language(s) used during the compilation process. However a machine independent root compiler will be made available for common use.

A compiler generator or code generator-generator program may be developed to speed production or adaptation of translators for all target computers.

7.3 Interfaces

7.3.1 Handling of Translators

A handling package shall be provided with each translator which permits the control of the execution of the various steps or phases of a translator, to select the input source, output target, terminate translation, repeat steps, etc.

This package has to operate either in an interactive mode or in batch, controlled by an appropriate job-control language (JCL). It is desirable that the control instructions and/or the functional capabilities of the handling package are standardized for DoD applications.

It shall be possible to limit acess to language features, which are particularly unsafe or error prone.

7.3.2 Input to Translators

7.3.2.1 Source statements (to be compiled)

The compiler must accept the standard input format as delivered by the editor.

7.3.2.2 Compiler control and option parameters

A translator shall accept target machine characteristics,such as:

Machine model,memory size,special hardware options,peripheral equipment, optional instruction sets available,libraries available (including runtime support).

The translator may accept this data either from the program input source, from direct query, or from a separate source.

Inputs to control the options of the translators shall include:

Listing controls,debugging controls (such as whether or not to output code for subscript-checking,assertion-checking,etc.),optimization options. The translator shall accept this information from the input source,from the machine specification or from a separate source.

7.3.3 Output of Translators

7.3.3.1 Code and Control information

The object code output of the compiler shall be formatted in accordance with the conventions for the standard link-loaders.

Aditionally there shall be information passed from the compiler to the run-time support routines for various purposes such as symbolic debugging, formatted dumps,error-checking,etc.

7.3.3.2 Listings

The compiler package ,through the combination of a set of appropiate modules, shall be capable of optionally producing at least the following listings :

Source as input to the compiler and before any conditional compilation statements are processed.

Source,but including,in the same format,any input statements retrieved from a source library by the compiler.

Source after all library retrievals and conditional compilation statements have been processed. [Note: all these listing formats shall permit the user to see statement identification or line number(for error displays),the occurence of errors,and the block level.]

Parallel listing of source and object code (when appropiate) with the same

options as the source-only listings.

A symbol attribute listing

A cross reference listing

A program structure map,which shall show the structure of the program with regards to blocks where data,procedure,function or path are declared. In addition,the map shall show any fetchs of definitions or input source from a library.

A compiler resource usage listing which shows the amount of computer resources used (Examples are : amount of computer time used,percentage and size of symbol table used)

A program resource usage estimate,showing e.g.the minimum execution time of procedures and processes,maximum buffer space,etc.

7.3.3.3 Error messages

Compilers shall be required to use a standard diagnostic and warning message format wherever applicable. All error messages shall be unambiguous. The implementor shall attempt to provide the following with each error: a plain language (as opposed to code number) description, the offending symbol or entity, and identification of the source context.

In case the error messages are mixed with the source and/or object listings, there shall also be an error summary listing giving a total count for each error and the line numbers on which that error occurred.

Chapter 8

Link/Load Tools

8.1 General Requirements

Link/Load tools shall be developed which are adapted to the special requirements of multiprocess systems. They should be able to link program-pieces which have been seperately compiled .These pieces may have been written in the Common Language or in other programming languages. Assembled code may be linked,too,but extensive checks shall be provided to try to minimize the inherent risks of this technique.

The link/load tools shall also check the type conventions as provided by the Common Language as well as other built-in protection mechanisms.They shall accept names of indefinite length.

Link/load facilities will be required making it possible to dynamically link new modules to existing systems. Some may to support maintenance by the inclusion of patching into the general link/load hanism.

8.2 Interfaces

The link/load tools shall have a standard input-interface which accepts the standard output of the respective compiler passes. In order to be able to link code which has been produced by a different compilation process,it may be necessary either to provide additional information to the link load tools or to apply an interface adaptation tool to the foreign code.

Additional control information may be necessary to completely control the link/load process which can either be provided by the translator's handling package or be contained in the output of the translation.

Besides program modules proper the link/load tools shall be able to accept as input mathematical packages,executive modules,I/O-routines,and the contents of pre-compiled application libraries.

As output the link/load tools shall provide error-messages,e.g. concerning non-matching inputs,and,optionally,memory maps which describe the final structure of the program after the link/load process.

Chapter 9

Runtime Tools

9.1 General

The necessary mathematical, I/O, and executive support routines shall be provided in the form of runtime support packages, wherever practical. These packages shall be written in a modular fashion, such that the support package which is actually required for a particular program can be generated at link/load time in order to reduce runtime overhead.

The runtime packages shall include routines which are necessary to interface the programs with runtime test and debug tools.

The runtime support routines may produce a summary of computer resources used in the execution of the program. An example is the amount of computer time and storage used.

9.2 Virtual Language Machine

A package shall be provided together with the translator which contains all necessary support routines for language elements which are neither directly compiled nor available on the respective target (virtual) machine. This package will either provide an adaptation to the existing operating system and runtime packages or will extend the capabilities of the bare machine in question to match the requirements of the language.

9.3 Extended Virtual Machine

Where practical, packages shall be provided, either with the compiler or from a separate source, which extend the capabilities of the language machine towards more powerful constructs, but are not application dependent in a strict sense.

Such packages will include:

Formatted-I/O
Graphic-I/O
Frequently used non-standard-I/O
Matrix calculation
Resource management routines, etc.

9.4 Runtime Test and Debug Tools

These runtime support packages shall be integrated with the runtime test and debug tools in such a way that the latter can refer to source code and work in an interactive way wherever practical.

The runtime displays for errors shall include the subprogram, definition module, or path, the procedure and the statement number on which the error occurred. The implementor shall attempt to display the offending symbol if any. The display shall also include a trace back of all currently executing or pending procedures, functions, paths, etc. A dump of all active variables may be at the option of the user.

The outputs of run-time debugging shall contain information similar to the error display.

The test and debug tools shall include the following:

9.4.1 Branch and Timing Counters

Methods of recording which branches of a program have been exercised shall be developed for testing. Frequency of execution and amount of CPU time required shall be given by the counter program. This program may be used to determine if all instructions in a program have been executed.

9.4.2 Trace and Breakpoint

Methods of recording the sequence of execution of instruc- tions or programs shall be developed for testing. The capability of stopping at prescribed instructions shall also be provided in order to examine or change the test conditions.

9.4.3 Interactive Symbolic Debugger

This tool shall allow inspection of the contents of variables, give information on the status of processes at the user's request, show the contents of queues, indicate resources used by a given process, etc.

Modifications to such entities should only be possible under the most stringent safety precautions.

9.4.4 Symbolic Dump

Methods of relating the results of tests (memory dumps) back to the source program shall be provided. The intent is to allow the programmer to debug his program in the common language rather than on assembly or machine code level.

Chapter 10

Maintenance

10.1 General

As maintenance has become the main cost-factor in the lifecycle of an automated system, all possible efforts shall be made to reduce this factor. Efforts to design the development tools in a way that maintenance is facilitated shall be undertaken. Additionally, the human factors of the work process itself as well as the properties of the environment, in which maintenance takes place, shall be investigated in order to derive improved methods, procedures and tools for maintenance.

10.2 Maintenance Oriented Precautions

Documentation shall be provided in a way, preferrably automatically, that it helps to facilitate maintenance in case it should be necessary. This shall hold for the design documentation as well as for all listings and supporting information which are created during the programming process proper.

The library system used for maintaining programs shall be capable of recording all changes to programs. Prompting may be used to encourage the programmer to identify information for understanding what was changed and why.

The translators shall pass as much symbolic information from source level to the object level as is practical in order to facilitate reference to the original information during maintenance.

10.3 Maintenance tools

Inverse assemblers and compilers as methods of deriving source code from object code shall be investigated to assist in maintaining code for which source is not available.

Incremental compilers, though typically test tools shall be investigated as to their potential of the handling of maintenance oriented changes.

A method of symbolically patching programs should be investigated, because it may help to increase the understanding of a change and to decrease the chance of error in making the change.

Chapter 11

Management Tools

11.1 General

Methods, standards and tools shall be developed which will make it possible to determine whether the resulting systems and programs are sufficiently bug-free and whether they meet the specified requirements as to functional capabilities as well as to time and space constraints. It is also desirable that this process be formalized and that the necessary control information be derived from earlier development stages.

Management should also have the possibility to restrict the use of potentially dangereous language features to certain persons or cases where they are safe to use.

11.2 Libraries

Application libraries, both on source-code and on link-load level shall be maintained in order to speed up program development and minimize duplication of effort. A library system shall be developed to facilitate retrieval of program modules. It is desirable that such a system is integrated into the design tools. The libraries shall also be capable of maintaining program specifications, program change histories and test cases.

11.3 Interface monitor

Programs to test interface specifications between modules within a software system shall be developed to assist in eliminating a major source of bugs.

Chapter 12

Application Software

One of the goals in the use of the Common Language is to increase portability of programs written in the Common Language. For those application programs written in the Common Language, portability will be promoted by the following methods.

1. Information concerning application programs will be maintained by the Language Support Agency and cataloged by type of program. A standard abstract format will be employed.

2. Major types of embedded systems will be identified and basic tasks within these types will be identified for catalog purposes.

Some types of embedded systems are:

a. Command and Control

b. Communications

c. Avionics

d. Shipboard

e. Test Equipment

f. Trainers and Simulators

g. Missile Guidance

h. Space Systems

i. Radar

j. Gun Control

k. Data Management

3. Organizations concerned with the Common Language will encourage special interest groups within user organizations to address the major types of embedded systems as well as common functions across all embedded systems.

Chapter 13

Training Support

Initial training will be required for programmers using the language, for

developers of compilers and support tools, and for management. Preparation of courses for each of the various levels will be required. Different modes of training will also be required due to diverse loca- tions, schedules, and background of those requiring training.

13.1 Types of Training Required

13.1.1 Programmers Using the Common Language

During the language introduction phase, training will be provided for all programmers who will write programs in the Common Language. Training will consider new as well as experienced programmers and will consist of beginning, intermediate and advanced levels.

Training will be required for language use as well as tool use. Language aspects which help accomplish project objectives such as reliability, efficient memory usage, efficient central processor usage, maintainability, and standard styles should be taught.

Training aids will include manuals for programmers familiar with other HOL's. For instance, documentation and courses may be required for programmers presently using Jovial, CMS-2, FORTRAN, etc. These must include not only the differences in the constructs of the languages, but also improved methodologies made possible by the use of the Common Language.

13.1.2 Compiler Developers

Training will be provided in the syntax and semantics of the language for personnel developing compilers. Experiences will be shared whenever possible.

13.1.3 Management of Projects Using the Common Language

The management of projects using the language will require overview training for the language and its environ- ment. Training in techniques which promote success in project development should also be prepared.

13.2 Training Modes

Methods of training shall include the following.

1. Classroom Instruction
2. Video Tape Courses
3. Computer Automated Instruction
4. Self-Instruction Manuals

Material for all of these methods shall include liberal use of programming examples with various levels of complexity and shall depict the required steps in arriving at a solution.

Materials will be provided in English. The extension of these materials to other languages, especially in the NATO community will be fostered and encouraged.

Chapter 14

Information Collection, Dissemination, and Promotion

It shall be the responsibility of the Language Support Facility (LSF) to collect and disseminate all information concerning the Common Language.

The LSF will maintain information about the Language as well as programs written in the language which support the language. This information will be cataloged in a hierarchical document which contains sections on all types of documents which pertain to the Common Language. The catalog will contain a brief description of each item of documentation in the form of a standard abstract. Each description will include title, purpose, author, revision level, size, and key words. The catalog will also include a Key-Word-In-Context (KWIC) listing for search purposes.

The LSF shall maintain statistical information about the use of the Common Language. Statistics shall include the number of projects using the Language, number of compilers, and number of computers for both host and target. Reports from the field must include information about the detail use of the language and compilers. The information should include error studies, difficult to use constructs, and amount of machine code used. These statistics shall be published periodically as part of a Common Language report. This report shall include the present status and plans for the language.

All of this information will be made available to the Common Language community to ensure that all users and potential users are working with accurate, current information.

A periodic bulletin may be distributed in published form and possibly computer accessible (i.e. on the ARPANET). The bulletin would contain information about the language and the environment.

ANHANG 6

CMU-CS-78-133

TARTAN

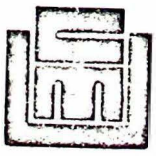
Language Design for the Ironman Requirement: Reference Manual

Mary Shaw
Paul Hilfinger
Wm. A. Wulf

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

June, 1978

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

CMU-CS-78-133

TARTAN

Language Design for the Ironman Requirement: Reference Manual

Mary Shaw
Paul Hilfinger
Wm. A. Wulf

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

June, 1978

Abstract: Tartan is an experiment in language design. The goal was to determine whether a "simple" language could meet substantially all of the Ironman requirement for a common high-order programming language.

We undertook this experiment because we believed that all the designs done in the first phase of the DOD effort were too large and too complex. We saw that complexity as a serious failure of the designs: excess complexity in a programming language can interfere with its use, even to the extent that any beneficial properties are of little consequence. We wanted to find out whether the requirements inherently lead to such complexity or whether a substantially simpler language would suffice.

Three ground rules drove the experiment. First, no more than two months -- April 1 to May 31 -- would be devoted to the project. Second, the language would meet all the Ironman requirements except for a few points at which it would anticipate Steelman requirements. Further, the language would contain no extra features unless they resulted in a simpler language. Third, simplicity would be the overriding objective.

The resulting language, Tartan, is based on all available information, including the designs already produced. The language definition is presented here; a companion report provides an overview of the language, a number of examples, and more expository explanations of some of the language features.

We believe that Tartan is a substantial improvement over the earlier designs, particularly in its simplicity. There is, of course, no objective measure of simplicity, but the syntax, the size of the definition, and the number of concepts required are all smaller in Tartan.

Moreover, Tartan substantially meets all of the Ironman requirement. (The exceptions lie in a few places where we anticipated Steelman requirements and where details are still missing from this report.) Thus, we believe that a simple language can meet the Ironman requirement. Tartan is an existence proof of that.

We must emphasize again that this effort is an experiment, not an attempt to compete with DOD contractors. Tartan is, however, an open challenge to the Phase II contractors: The language can be at least this simple! Can you do better?

This work was supported by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074 (monitored by the Air Force Office of Scientific Research).

1. Basic Concepts and Philosophy 1
2. Basic Structures 3

2.1. Primitive Expressions 3
2.2. Identifiers 4
2.3. Lexical Considerations 5

3. Expressions 6

3.1. Invocations 7
3.2. Dynamic Allocation 8

4. Statements 9

4.1. Blocks 9
4.2. Sequenced Statements 9
4.3. Assignment Statement 9
4.4. Conditional Statements 10
4.5. Loop Statements 11
4.6. Unconditional Control Transfer 11
4.7. Exceptions 11
4.8. Parallel Process Control 12

5. Types 13

5.1. Scalar Types 13
5.2. Composite Structures 13
5.3. Dynamic Types 14
5.4. Process Control Types 14
5.5. Defined Types 14

6. Definitions and Declarations 15

6.1. Declarations 15
6.2. Modules 16
6.3. Routines 16
6.4. Exceptions 16
6.5. Type Definitions 17
6.6. Generic Definitions 17
6.7. Translation Issues 18

I. Standard Definitions 19

I.1. System-Dependent Characteristics 19
I.2. Properties of Types 19
I.2.1. Fixed 19
I.2.2. Float 19
I.2.3. Enumerations 19
I.2.4. Boolean 20
I.2.5. Characters 20
I.2.6. Latches 20
I.2.7. Arrays 20
I.2.8. Sets 20
I.2.9. Dynamic Types 20
I.2.10. Records 20
I.2.11. Variants 21
I.2.12. Strings 21
I.2.13. Activations 21
I.2.14. Actnames 21
I.2.15. Files 21
I.3. Alphabets 21

II. Collected Syntax 22

1. Basic Concepts and Philosophy

A program is a piece of text that describes a sequence of actions intended to effect a computation. The process of "executing a program" to obtain this effect is called elaboration of the text.[1]

Programming languages are used for communicating programs, both between people and between people and machines. Although the program text is static, the concepts being communicated are dynamic. This dynamic nature of a computation can make it difficult to communicate the ideas underlying a program, and especially to communicate these ideas between people. To expedite the communication, we impose structure on the way languages are used. Although this structure restricts what can be written, it results in regular patterns for expressing decisions. The human reader benefits from this by developing expectations about how these ideas will be expressed.

Programming languages encourage the imposition of structure by providing notations for the structures whose use their designers wish to promote. During the process of language design, our beliefs about programming methodology and the state of language processing technology lead us to formulate concepts and structural rules. We then select syntactic forms and structuring features to emphasize these concepts. We expect that this will simplify the task of describing programs with the attributes we view as "good structure" and that programmers will, as a result, be encouraged to organize their programs this way.

We distinguish three dominant structures in Tartan programs: (1) the lexical structure, which organizes the static program text, (2) the control structure, which organizes the dynamic execution, and (3) the data structure, which organizes the information on which computations are performed.

- Lexical structure is a property of the program text. Programs are divided hierarchically into sections, called lexical scopes, that share information about data. Scope determines the interpretation of identifiers, so all the text in a given lexical scope shares the same vocabulary -- definitions, variables, etc. Scope rules permit some identifiers to be used with the same interpretation in several lexical scopes.

- The control structure of the program determines the order in which its statements are executed.

- The structure imposed on data involves the concepts of type, values, and variables. Ultimately, computations are performed on values; we take that notion to be primitive: values exist, and each has exactly one type, which determines the legal operations on the value. Values are stored in variables, which are objects produced by elaborating type definitions. Variables, too, have types; these types determine the sets of values that may legally be stored in the variables.

These fundamental structures interact in a number of ways. Two major interactions appear as the concepts of extent and binding. The control and lexical structures interact to determine extent. The extent of a variable is its lifetime -- the time during which it affects or is affected by the elaboration of the program. Binding rules are invoked by both lexical and control structures; they associate identifiers with program entities (objects, modules, routines, types, labels, and exceptions).

In Tartan, programs are composed of definitions, declarations, and executable statements. A definition binds an identifier to a module, routine (procedure, function, or process), type, or exception; it is processed during translation. A declaration binds an identifier to an object (i.e., a variable or value); it is processed at run time, usually to allocate storage. Executable statements are elaborated at run time to effect actual computations -- manipulation of values.

Lexical structure is imposed on Tartan programs by blocks and modules, which delimit lexical scopes. These scopes may be nested arbitrarily. Both constructs may use identifiers defined in other scopes; both may define identifiers that can be used in other scopes. Blocks and modules differ only

---

[1] We use the word "elaboration", in preference to "execution", to connote actions taken during translation as well as during execution. Elaboration may be thought of as an idealized, direct execution of the textual version of the program.

in their scope rules and in their effects on the extent of variables. Tartan has two scope rules:

- An **open** scope **inherits** (imports automatically) all the identifiers that are defined in its enclosing scope. It may not export any identifiers. Blocks are open scopes except when used as routine bodies.

- A **closed** scope inherits all identifiers that are defined in its enclosing scope except those for labels and nonmanifest objects.[1] It may explicitly import identifiers for objects, provided they have global extent. All modules are closed scopes, as are blocks when they are used as routine bodies. A closed scope that is a module may **export** identifiers that name variables, modules, routines, types, or exceptions.

Identifiers that are exported from an inner scope or imported from an outer scope have the status of identifiers defined in the scope. Redefinition of identifiers within a scope is not permitted; however, this does not prohibit **overloading** of routine names. In addition, the same identifier may be imported with different meanings from two different scopes. Such identifiers are qualified with the names of the modules in which they were defined; thus they are not duplicate definitions. Similarly, literals and constructors are qualified with their types to prevent ambiguity. In either case, the module or type qualifier may be omitted if no ambiguity arises.

In Tartan, extent is controlled exclusively by blocks. Only instantiated objects (variables, constants) have extent. Variables are instantiated by the elaboration of declarations (for named variables) and by **explicit construction** of variables having **dynamic types** (dynamically created variables). Named variables have extent coincident with the surrounding block. Dynamically created variables have extent coincident with the block containing the definitions of their dynamic types. Formal parameters of routines are considered to have extent coincident with the routine body.

Tartan provides a facility for making **generic definitions** of routines and modules. This allows the programmer to write a single textual definition that serves as an abbreviation for many closely-related specific definitions. The definitions may accept parameters; the parameters are completely processed during translation. The effect of using a generic definition is that of lexically substituting the definition in the program at the point of use.

The syntactic definition of Tartan uses conventional BNF with the following additions and conventions:

- Key words (reserved words) and symbols are denoted with **boldface**.

- Metasymbols are denoted by lower-case letters enclosed in angular brackets, e.g., "<stmt>".

- The symbols { and } (not in boldface) are meta-brackets and are used to group constructs in the meta-notation.

- Three superscript characters, possibly in combination with a subscript character, are used to denote the repetition of a construct (or a group of constructs enclosed in {}):
  "*" denotes "zero or more repetitions of"
  "+" denotes "one or more repetitions of"
  "#" denotes "precisely zero or one instance of".
  Since it is often convenient to denote lists of things that are separated by some single punctuation mark, we denote this by placing the punctuation mark directly below the repetition character.

The semantics of the language are described in English. In the interest of a compact and regular syntax, we have allowed syntactic constructs that are disallowed on semantic grounds. This is consistent with standard practice with respect to, for example, undeclared identifiers.

---

[1] Literals and identifiers for variables that are declared **manifest** are manifest objects; hence they are inherited.

2. Basic Structures

2.1. Primitive Expressions

```
<const>       ::= <digit>+ { . <digit>+ }* | true | false | nil | closed | open | mint | empty
                | <constructor> | <id> | <qual id> ' <const> | <type> ' <const> | <expr>
<constructor> ::= ( <expr>,* ) | ( { <option> -> <expr> },+ ) | " <char>* "
```

Some examples are:

```
123.456
Color'green
true
Person'("Sam",21,male)
"efg"
(1..2->0.1, 3..4->0.5, others->1.0)
```

Primitive expressions form the basis for the recursive defintion of expressions. They are the elements referred to as constants, literals, and constructors in programming languages and as generators in algebras.

Constants and literals denote values. The type of a constant is determined by its declaration. The types of literals are determined as follows:

- A sequence of digits containing no decimal point is of type Int. Type Int is defined in terms of type fixed for each machine as described in Appendix I.1.

- A sequence of digits containing a decimal point is of type Real. Type Real is defined in terms of type float for each machine as described in Appendix I.1.

- If a sequence of digits, with or without a decimal point, is qualified by a fixed or float type or by a defined type that is ultimately defined in terms of fixed or float, the type of the literal is determined by the qualifier.

- True and false denote boolean values. Nil denotes the null value for any dynamic type. Open and closed denote values for latches. Empty denotes the empty set. Mint denotes an activation of any process in mint state.

- A character string containing one character is a literal of type char. Any other character string is a constructor of type string.

Literals and manifest expressions are evaluated during translation with the same algorithms and accuracy as are used during execution.

If an <id> is to be a <const>, it must have been declared const or be a member of an enumerated type. If an <expr> is to be a <const>, it must be a manifest expression.

The type of a constructor may be indicated by a prefixed qualifier. If the qualifier is omitted, the constructor is assumed to give the value of an array indexed with integers beginning at 1. Constructors are provided for composite and dynamic types.

- If the constructor has a record type, the <expr>s in parentheses give the field values in the order of their declaration.

- If the constructor has an array type, the parenthesized list gives the element values. If the constructor is a simple expression list, it gives the values in order from lowest index to highest. If the constructor uses the form with options, the expressions in the <option>s indicate the array position to which each value corresponds. The special constant others may appear as the last <option>; it will match any constant that is not included in any other <option>. The constructor form with options is legal only for arrays and for types ultimately defined in terms of arrays; the expressions in the <option>s must be manifest.

- If the constructor has a variant type, the first expression in the parenthesized list is the tag and the remainder of the list is a constructor for the corresponding variant.

- If the constructor has dynamic type, the result is a pointer to a new variable having the attributes supplied in the type qualifier and the value given by the parenthesized list.

A constructor containing no <expr> provides an uninitialized instance of the indicated type.

2.2. Identifiers

```
<var id>    ::= <qual id> | <var id> ( <actuals> ) | <var id> . <id> | <var id> ( <range> ) | Rep' <id>
<range>     ::= <expr> . . <expr> | <type>
<option>    ::= { <const> | <range> },*
<qual id>   ::= { <id> '}* <id>
<id>        ::= <letter> <letter or _ or digit>*
```

Some examples are:

```
Animal'Cat
V(3)
V(1..N)
Sam.Age
Ident_with_mark
```

Identifiers have no inherent meanings. They are associated with objects, routines, modules, types, statements, and exceptions. Declarations and definitions establish the meanings of identifiers within particular scopes.

Identifiers may be simple, or they may be qualified with module or type names in order to resolve ambiguity among names exported from several modules.

Identifiers that name objects are <var id>s. They may be simple identifiers, they may be qualified to indicate where they were defined, or they may name elements or substructures of composite structures.

- Simple <var id>s (i.e., <qual id>s used as <var id>s) are identifiers declared in variable declarations or by the <formals> in a routine header.

- The form <var id>(<actuals>), where <var id> denotes an array, denotes the element of that array indexed by the <actual>s. The types of the actuals must match the index types for the array.[1]

- The form <var id>(<actuals>), where <var id> denotes a variable of a variant type and the <actual>s consist of a single <expr>, indicates that the tag field of the <var id> must be <expr> and denotes the value of that option of the variant type. On the left side of an assignment, this form has the effect of setting the tag field; the expression on the right side of the assignment must be of compatible type.

- The form <var id>(<range>) denotes a subarray. The <var id> must denote an array and the limits of the <range> must match the declared type of the array's index set and be a subrange of the declared range. The subarray consists of the indicated elements of the <var id>, in the same order as they appear in the <var id>. If the index type of the array is fixed or defined in terms of fixed, the subarray is indexed by integers beginning with 1; otherwise it is indexed from the minimum value of the index set of the array.

- If <var id> denotes a record object, the form <var id>.<id> denotes the field named <id> in that record object. If <var id> denotes an object of dynamic type, then <var id>.<id> denotes the field named <id> in the record object pointed to by the value of <var id>; <var id> must not have the value nil. This form is also used to access the value of a variant tag or the attributes associated with the type of a value or variable. In addition, if T is a variable of dynamic type, T.all is the complete value (all components) of the object associated with T.

---

[1] Note that the index types include range restrictions.

- The form Rep'<id> is used in the same scope as the definition of the <id>'s type to indicate that the <id> is to be regarding as having the underlying type. This permits operations on the underlying type to be used for defining operations on the new type.

Identifiers that refer to definitions (e.g., of routines, types, or modules) are <qual id>s.

When an identifier is exported from a module, in the scope to which it is exported it is referred to by a <qual id> or <var id> constructed by prefixing the identifier with the name of the module from which it is exported. The qualifier is separated from the identifier with an apostrophe. Qualifiers may be omitted if no ambiguity results.

A <type> used as a range must be fixed, an enumerated type, or a defined type that is ultimately defined in terms of fixed or an enumeration.

2.3. Lexical Considerations

Spaces may be inserted freely between lexemes without altering the meaning of the program. An end-of-line is equivalent to a space and may not be part of a lexeme. At least one space must appear between any two adjacent lexemes composed of letters, digits, underbar, and decimal points. In identifiers, all characters are significant, but alphabetic case is not.

Comments are introduced by the character "!" and terminated by the next following end-of-line. They have no effect on the elaboration of the program.

Although the language as presented in this report takes advantage of characters that are not in the 64-character ASCII subset, simple substitution to map programs into that alphabet are defined in Appendix I.

3. Expressions

```
<expr>       ::= <unop>* <var id> | <unop>* <const> | <unop>* <func call>
                 | <unop>* ( <expr> ) | ( <expr> ) . <id> | <expr> <binop> <expr>
<unop>       ::= - | ¬
<binop>      ::= * | / | + | - | < | ≤ | > | ≥ | = | ≠ | ∧ | cand | ∨ | cor | ↑
<func call>  ::= <qual id> ( <actuals> )
<actuals>    ::= <expr>,*
```

Some examples are:

```
x
x + y
sin(x)
-(x*y + z*w)
(Root.Ptr).all
```

Expressions describe computations that yield values. The elaboration of an expression produces an object containing the value of the expression. The type of the object is determined by the following rules:

- The type of an <expr> that is a <var id>, <const>, <func call>, or selection of a field from a computed composite value is determined by the appropriate declaration (or rule for literals).

- The type of a parenthesized expression is the type of the expression inside the parentheses.

- The type of a binary infix expression or a unary expression is determined by the definition of the appropriate binary or unary operator definition. These operators represent invocations of functions that may be overloaded. The appropriate operator definition must therefore be determined on the basis of the types of the operands.

The usual operations are associated with the operators +, -, *, /, ↑, ¬, ∧, ∨, cand, cor, <, ≤, ≥, >, =, and ≠. The programmer may overload these function names, but the added definitions must be unary or binary to conform to the established syntax. Precedence rules for the unary and binary operators are given by the following table, in which operators on a single line have the same precedence and operators higher in the table bind more tightly than operators lower in the table. Unary operators have the highest precedence.

```
¬  -
↑
*  /
+  -
<  ≤  ≥  >  =  ≠
∧  cand
∨  cor
```

Within precedence levels, associativity is left-to-right.

For all operators except cand and cor, elaboration of an expression proceeds as if the expression were written in functional form (see section 3.1). For cand and cor, the left operand is elaborated first and the right operand is elaborated only if necessary.

A manifest expression is a literal, a value of an enumeration type, an identifier declared with manifest binding, a generic parameter, a manifest type attribute, a constructor involving only manifest expressions, or any expression involving only these expressions and language-defined operations. The value of a manifest expression is known during translation.

3.1. Invocations

Some examples are:

```
F(5)
Sequence'Insert(S,5)
P()
```

An invocation causes the elaboration of a procedure or function body with the elements of the <formals> list of the routine bound to the elements of the <actuals> list provided by the invocation. If a routine name is overloaded, the definition whose formal parameter types match the types of the actual parameters is selected. Procedure and function invocations (<proc call> and <func call>) differ in that procedure invocations are statements, whereas function invocations are expressions having values. An invocation consists of the following steps:

- Elaborate each of the <actuals> in an unspecified order, yielding a sequence of objects.
- For each result formal, create a variable having the same type and attributes as the corresponding actual. Bind the result formals to these variables.
- For each const or manifest formal, create an object of the specified type with the same attributes as the corresponding actual. Copy the value of the actual into the new object.[1]
- Bind each var formal to the corresponding actual, which must be a variable (i.e., a <var id>). Thus var formals are passed by reference.
- With the bindings established, elaborate the body of the routine.
- For each result formal, copy the final value of the variable bound to that formal back into the corresponding actual, which must be a variable (i.e., a <var id>). Note that this actual is determined before the elaboration of the routine (i.e., for the actual A(i), it is the initial and not the final value of i that determines the variable that receives the result).

The result of a function is treated as a result parameter instantiated at the call site with extent as described above and passed as an implicit parameter to the function. During the elaboration of the function, its value is developed in this result parameter.

During elaboration of a function, assignment to a variable that is not local to the function body (or to the body of a routine it invokes, directly or indirectly) is permitted only if the function is never invoked in a scope where such a change is made to a variable or component that is directly accessible by the caller.

Actual parameters are matched with formal parameters positionally. They must satisfy restrictions on type, binding and aliasing.

- The type of an actual parameter is acceptable if its <type name> exactly matches the <type name> of the corresponding formal parameter. Type attributes (instantiation parameters of a type) play no role in type checking. Chapter 5 gives rules for determining <type name>s.
- The binding of the actual parameter is acceptable if it matches the <binding> of the corresponding formal parameter according to the following rules:

If the formal parameter is	then the actual parameter may be
var	<var id> declared var
const	<expr>
manifest	any manifest <expr>
result	<var id>

- Finally, the set of actual parameters must satisfy the following nonaliasing restriction: A variable may not be used in more than one var or result position of a single procedure or

[1]Note that for dynamic types, this is a pointer copy.

process call. For the purpose of testing this restriction, imported variables are considered to be actual parameters bound as specified in the import list.

3.2. Dynamic Allocation

Each use of the constructor for a dynamic type creates a distinct element of the type. Each such element remains allocated as long as there is an access path to it.

Attributes of the dynamic type are provided when the constructor is used. Thus it is possible to associate objects with different attributes with the same dynamic variable at different times.

4. Statements

```
<stmt>        ::= <proc call> | <id> : <stmt> | <empty> | <block>
                | <var id> := <expr>
                | if <expr> then <stmt>;* { elif <expr> then <stmt>;* }* { else <stmt>;* }* fi
                | case <expr> { on <option> -> <stmt>;* }+ esac
                | while <expr> do <stmt>;* od | for <id> in <range> do <stmt>;* od
                | goto <id>
                | signal <qual id> | resignal | assert <expr>
                | <stmt> { { on <id>,+ -> <stmt>;* }+ }
                | create <var id> ( <actuals> )
<proc call>   ::= <qual id> ( <actuals> )
<block>       ::= <code body>
<code body>   ::= begin { <def-decl> ; }* <stmt>;* end
```

Statements designate actions to be performed. Their elaboration results in changes in the execution state of the program. The <empty> statement has no effect. Labels are used by **goto** statements in altering the flow of control in a program. A label is accessible only within the <stmt> it labels and within a compound statement (sequence of <stmt>s separated by semicolons) of which it is a <stmt>.

4.1. Blocks

Some examples are:

```
begin var x: boolean; x := true end
begin x := y; y := z; end
```

Blocks control extent. A <block> is elaborated when control flows into it, either because the <block> is the body of a routine that has been invoked or because the elaboration of another <stmt> has transferred control to it. First, all declarations and the texts of all module definitions are elaborated, in lexical order. Next, the <stmt>s are elaborated as described elsewhere in this chapter. Finally, the <block> is exited or terminated. If it is exited, control waits for all activations declared in this <block> to become dead or mint, then the extent defined by the <block> is closed and all nondynamic variables instantiated in the <block> are deallocated. If the <block> is terminated, all activations declared in the <block> are forcibly terminated, and then the <block> is exited. The choice between exiting and terminating the block depends on how control arrived at the end of the block. If the block came to an end because a handler completed or an enclosing process was terminated, the block is terminated. Otherwise, it is exited.

A <block> is not permitted to export identifiers. Except when used as a routine body, it is an open scope and has no need to import any.

4.2. Sequenced Statements

Some examples are:

```
x := 1; y := 2; z := 3
SumSq := 0; for i in 1..10 do SumSq := SumSq + V(i)↑2 od
```

Sequenced statements are elaborated in the order given, except when that order is interrupted by a goto or an exception.

4.3. Assignment Statement

Some examples are:

```
V(5).Sum := 0
x := (3 + u) * y
```

The assignment statement "V := E" is a procedure call on an appropriate assignment operator, defined

```
proc ":="(var LHS:T, const RHS:T)
```

for arbitrary type T. The value of the second parameter is assigned to the object named by the first parameter. The parameters are of the same type, and the normal type-checking rules apply.

Assignment operators are defined for all primitive types. Assignment operators are defined for arrays, records, variants, and programmer-defined types if and only if they have no components that are declared const or are nonassignable by virtue of this rule. An assignment operator that copies the whole value is automatically supplied for each user-defined type. For dynamic types this is a pointer copy. Although assignment may be invoked with any variable and value of the type, it requires that the attributes of its left and right operands be identical, and signals the BadAssign exception if they are not. The BadAssign exception is also signalled if an assignment involving mismatched array, string, or set sizes or an activation not in mint state is attempted.

4.4. Conditional Statements

Some examples are:

```
if A < 3 then x := y fi
if x ~= 0 cand y/x > 0 then z := sqrt(y/x) else w := 1.0; q := y/x fi
case Tint
    on fuchsia -> Hue := cool; Description := "Purplish-red"
    on puce -> Hue := warm; Description := "Brownish-purple"
    esac
```

In the statement "if B then S1 else S2 fi", B must have type boolean. First, B is elaborated. If the resulting value is true, S1 is elaborated; otherwise S2 is elaborated. In the absence of an else clause, S2 is taken to be the empty statement, which has no effect.

The expression

```
if B1 then S1 elif B2 then S2 ... elif Bn then Sn else Sw fi
```

is equivalent to

```
if B1 then S1 else
    if B2 then S2 else
        . . .
            if Bn then Sn else Sw fi
        . . .
    fi
fi
```

In the statement

```
case E0
    on E11,...,E1k -> S1
    on E21,...,E2l -> S2
        . . .
    on En1,...,Enm -> Sn
    on others -> Sw
    esac
```

The E's must all be expressions of the same type, and all except E0 must be manifest. The type of the E's must be fixed, an enumerated type, or a defined type that is ultimately defined in terms of fixed or an enumeration. Any of the E's except E0 may be a <range>; such an Eij is treated as the sequence of values in the range. First, E0 is elaborated. The Eij are elaborated and the results are compared to E0 (in unspecified order). If E0 is equal to some Eij, the corresponding Si is elaborated. If all comparisons yield false, Sw is elaborated. Exactly one Si is elaborated for each correct elaboration of the case statement. If the special constant others does not appear as the last <option> and no match is found, an exception (CaseFailed) is signalled.

4.5. Loop Statements

Some examples are:

```
while x < 2.5 do x := F(y,x); y := G(y,x) od
for i in 1..10 do V(i) := i od
for hue in color do Tint(hue) := hue od
```

The loop while E do S od repeatedly elaborates if E then S fi until E becomes false. If E is initially false, the loop has no effect (other than the possible hidden effects or exceptions caused by the elaboration of E.)

The for statement for i in R do S od repeats the steps

- Bind i (as a constant) to a value in the range R.
- Elaborate S.

once for each element of the range R, in order. If R has no elements, the loop has no effect. The scope of the loop constant is restricted to the loop.

4.6. Unconditional Control Transfer

An example is:

```
goto L
```

The effect of a goto statement is to force control to the beginning of the statement with the given label. Since the scope rules prevent inheritance of labels across closed scope boundaries and importation of labels, a goto can not be used to transfer out of a routine or module.

4.7. Exceptions

Some examples are:

```
signal TooBig
assert x < 0

read(file,x) { on EOF -> goto Exit }
x := x+1 { on Overflow -> x := 0 }
```

Exceptions are processed by handler clauses associated with individual statements. Each handler clause associates processing code with given exceptions. The special identifier others may appear as the last <id> list of a handler clause; it matches any exception that is not named in some other exception <id> list of the same clause.

When an exception is signalled, control is transferred to the nearest dynamically enclosing handler clause that handles the exception, either explicitly or through an others clause; the elaboration of the handler replaces the elaboration of the remainder of the statement. If this handler is not in the currently-executing block, all intervening blocks will be terminated. If a handler is not found within the currently-executing routine, that routine is terminated and the exception is resignalled at the point of call of the routine. If a handler is not found within the currently-executing process, that process is terminated and the exception is resignalled at the end of the block in which the process activation was declared after waiting for control to reach that point and for all other activations declared in that block to terminate. If no handler is found in the scope of the exception name, a default handler will be supplied to terminate that block.

Exiting a handler causes termination of the <stmt> with which it is associated. If the handler resignals the same exception or raises a new one, the normal rules for exception processing apply.

The resignal command may be used in any handler body to resend the signal that caused that handler to be invoked.

The assert statement raises the assertion exception if the <expr> is false. It is exactly equivalent to the statement "if ¬ <expr> then signal assertion fi".

There is one exception to the rule that an exception must be handled by the block in which it is signalled or by a caller of that block: the Notify operation on activations or actnames. The effect of a Notify is as if the Terminate exception were signalled in the currently-executing statement of the activation named by the Notify command.

4.8. Parallel Process Control

Some examples are:

```
create P(S)
activate(P1)
if IsBlocked(P1) then . . .
```

The create command instantiates a process, P, as an object of type activation-of-P. The <var id> in a create must name an object of type activation-of-P that is in mint state. If a process takes any var parameters, the corresponding actual parameters must have extent at least as great as the activation variable. The effect of the create is to instantiate an activation of P, bind the actuals of the create to the formals of P, and set the activation in suspended state.

Each activation has a unique identifying token value of type actname, and it may be named by one or more objects of type actname. Except for create, all operations that control parallelism are special routines that operate on either actnames or activations. These routines control the processes and parallelism by changing and interrogating the states of individual activations; they are described in Appendix I.2.

Note that the extent rules require an activation to be dead or mint before the block in which it is declared can be exited. This provides an implicit join operation. A fork can be obtained with a series of creates and activates.

5. Types

```
<type>       ::= fixed( <actuals> ) | float( <actuals> ) | boolean | latch | char | file( <actuals> )
                 | enum[ <id>,+ ] | enum[ { " <char> " },+ ] | <expr> . . <expr>
                 | set( <actuals> ) | string( <actuals> )
                 | array ( <range>,+ ) of <type> | record [ <declaration>,+ ]
                 | variant <declaration> [ { on <option> -> <type> }+]
                 | dynamic <type> | activation of <qual id> | actname
                 | <type name> { ( <actuals> ) }*
<type name>  ::= fixed | float | boolean | latch | char | file | set | string
                 | enum[ <id>,+ ] | enum[ { " <char> " },+ ]
                 | array [ <type name>,+ ] of <type name> | record [ { <id>,+ : <type name> },+ ]
                 | variant [ <type name> { on <option> -> <type name> }* ]
                 | dynamic <type name> | activation [ <qual id> ] | actname
                 | <qual id> { [ <qual id>,+ ] }*
```

In Tartan, a <type name> may be either a simple identifier or an identifier inflected with additional type names. The <type name> so formed captures all the information needed for type checking.

- The <type name>s for the primitive scalar and simple nonscalar types are the keywords used to declare them: fixed, float, boolean, latch, char, set, string, actname, file.

- The <type name> for an array declared "array(a..b) of D" is "array[I,D]", where I is the <type name> of a and b.

- The <type name> for an enumeration declared enum[L1,L2,...Ln] is enum[L1,L2,...,Ln].

- The <type name> for an activation declared activation of P is activation[P].

- The <type name> for a dynamic type declared dynamic T is dynamic T.

- The <type name> for a record type is based on the sequence of field names and <type name>s in its declaration. For a record declared "record[F1:T1, F2:T2, ..., Fn:Tn]" the <type name> is "record[F1:TN1, F2:TN2, ..., Fn:TNn]", where the Fi are lists of field names, the Ti are types, and the TNi are type names. Bindings in the declaration do not appear in the type name.

- The <type name> for a variant is "variant[TT,T1->V1,T2->V2,...,Tn->Vn]", where TT is the <type name> of the tag, Ti is the i^{th} value of the tag type, and Vi is the <type name> that corresponds to the i^{th} value of the tag type. As a result, two variant <type>s are the same if they specify the same <type>s for all values of the tag.

- The <type name> for a defined type is the name given in the type definition.

5.1. Scalar Types

Some examples are:

```
Real
1..10
enum[fuchsia, ochre, puce, saffron]
```

Built-in scalar types include fixed, float, boolean, latch, and character. Integer and real must be constructed as special cases of fixed and float. Ordered scalar enumerated types are defined by providing an ordered list of values.

Types fixed and float require <actuals> lists to provide range, scale, and precision when they are used in declarations. These are attributes and do not affect the type. Although bindings for attributes may in general be const or manifest, the specifications of fixed and float require manifest attributes.

To define a type, the <expr>s in an explicit range must be const or manifest.

5.2. Composite Structures

Some examples are:

```
array(1..10) of Color
array(Color) of Real
string(10)
record[Name:string(35), Age:Int]
variant b:boolean [on true -> Int on false -> char]
```

Nonscalar data structures may be built up in three ways: with arrays (homogeneous indexed linear structure), with records (nonhomogeneous structures with named fields), and with variants (structures whose substructure may vary with time). In addition, the nonscalar types set, string, and file are defined.

Legal bindings for fields of records and variants are var, const, and manifest. If a <binding> is empty, it is taken to be var.

A variant type must have exactly one tag field. The special constant others may appear as the last <option> of a <variant type>; it matches any constant that does not appear in any other <option>.

The syntax for arrays provides an abbreviation for a set of types pre-defined as "array[IxType,EltType](r)" where IxType is the index type, EltType is the element type, and r is a (sub)range of IxType. Thus "array(1..10) of float" means "array[int,float](1..10)". Its type name, "array[int,float]", is written "array[int] of float". As for any type, when an <array type> is used as a formal parameter, the attributes are not supplied. The type "array(A,B) of T" is an abbreviation for "array(A) of array(B) of T". Similarly, the array accessor "V(i,j)" is an abbreviation for "V(i)(j)".

5.3. Dynamic Types

Some examples are:

```
dynamic Real
dynamic record [Data: Int, Next: ListElt, const Index: Int := K]
```

Values of a dynamic type are pointers to variables whose structure corresponds to the type definition. They are initialized to nil. The extent of these variables covers the entire scope of the type definition. Elaborating a constructor for the dynamic type yields a pointer to a new variable distinct from all others. The constructor supplies the attributes for this variable; they are not supplied in the declaration of the named variable of the dynamic type. Thus a named variable of dynamic type may at different times point to several different variables having different attributes.

5.4. Process Control Types

Some examples are:

```
activation of P
actname
```

Parallel processes are controlled with data of two types -- activations of processes and actnames, or names of activations. Activations are instantiations of a given process; an activation may contain at most one process activation during its lifetime and then only of the process given in its <type>. An actname value is a pointer to an activation. Actname variables may contain pointers to activations of any processes; an actname variable may refer to different instantiations of different processes from time to time.

An activation is used to control parallel or pseudo-parallel execution of a process. At any time it may be in one of four states: mint, active, suspended, and dead. The extent of an activation variable coincides with its scope. The immediately enclosing block cannot be exited until all activations declared within it are dead or mint. An activation is associated with exactly one process, which must be named by the <qual id>.

An actname may refer to any instantiated process. A newly-declared actname or activation variable is initialized to mint.

5.5. Defined Types

Some examples are:

```
T(n)
Sequence[int](50)
```

Programmers may define new types. See section 6.5 on Type Definitions.

6. Definitions and Declarations

```
<def-decl>      ::= <declaration> | <mod def> | <routine def> | <type def> | <generic def> | <empty>
                  | imports <qual id>,+ | exports <qual id>,+ | exception <id>,+ | disable <id>,+
                  | prag <proc call>;+ ;* garp
<declaration>   ::= <binding> { <id>,* { : <type> }* { := <expr> }* },+ | <binding> { <id>,+ : <type name> },+
<mod def>       ::= module <id> <mod text>
<mod text>      ::= ; <code body> | <remote inst>
<routine def>   ::= proc <id> <proc text> | func <id> <func text> | process <id> <proc text>
                  | func " { <unop> | <binop> } " <func text>
<func text>     ::= ( <formals> ) <id> : <type> ; <block> | <remote inst>
<proc text>     ::= ( <formals> ); <block> | <remote inst>
<type def>      ::= type <type name> { ( <formals> ) }* = <type>
<generic def>   ::= generic module <id> [ <formals> ] <mod text> | generic func <id> [ <formals> ] <func text>
                  | generic proc <id> [ <formals> ] <proc text> | generic process <id> [ <formals> ] <proc text>
<remote inst>   ::= is <qual id> [ <actuals> ] | is assumed ( <id> )
<formals>       ::= { <binding> <id>,+ : <type name> },*
<binding>       ::= <empty> | var | const | manifest | result
```

6.1. Declarations

Some examples are:

```
var     x: Real
const   y:= true
var     Hue1, Hue2, Hue3: Color
var     Tint := enum[saffron, puce, fuchsia, ochre]
var     V: array(5..7) of Int
var     M1:Mark(5), M2:Mark(7)
manifest   PI: Real := 3.14
```

The syntax for declarations allows three kinds of abbreviations. If the initialization expression appears, the type of the variable is evident from the <expr> and the ":<type>" may be omitted. In addition, lists of <id>s with the same types or bindings may be condensed. These abbreviations are illustrated by the following five declarations, all of which have the same effect:

```
var x,y := 0
var x,y:Int := 0
var x := 0, y := 0
var x:Int := 0, y:Int := 0
var x:Int := 0; var y:Int := 0
```

Elaboration of a declaration causes instantiation of an object which is the variable. Each variable has a type and a value. The type is determined when it is instantiated, but the value may be changed by further elaboration of the program. A variable may be restricted to be **const** (value fixed at block entry) or **manifest** (value fixed during translation).

Elaboration of a declaration proceeds as follows:

- Evaluate the <expr>, if present. It must be present in **manifest** or **const** declarations. It must be **manifest** in **manifest** declarations.
- If the <binding> is **manifest**, bind the value of the <expr> to the identifier(s).
- If the <binding> is **const** or **var**, elaborate any <actual>s in the <type> and instantiate a new variable with the indicated type and attributes for each identifier. If there was an <expr>, assign its value to each of the new variables.

When the type is dynamic, the declaration supplies the <type name> only (no attributes). In this case, only the pointer is allocated at block entry; the attributes are supplied when the dynamic type is actually (dynamically) allocated.

6.2. Modules

An example is:

```
module CounterDef;
    begin
    exports Counter, Reset, Incr, Value;
    type Counter = Int;
    proc Reset(result C:Counter); begin C := 0 end;
    proc Incr(var C:Counter); begin C := C + 1 end;
    func Value(const C:Counter)x:Counter; begin x := C end
    end
```

The elaboration of a module takes place during the elaboration of declarations for the block in which the module is defined. This elaboration consists of elaborating the declarations of the module in lexical order, then elaborating the statements of the module.

A module or routine inherits identifiers for definitions (modules, routines, exceptions, and types) from its enclosing scope. It may explicitly import identifiers of objects from that scope, provided the objects have global extent. A module, but not a routine, may export definition and object identifiers to its enclosing scope. Types, named routines, field accessors for records, and variables are exported by including their names in the exports list of the module. The right to apply infix operators, constructors, subscripts, ".all", or the create command for a type T are exported by including the special names T'infix, T'constr, T'subscr, T'all, and T'create, respectively, in the exports list. Literals of enumerated types are exported automatically if the types are exported.

6.3. Routines

Some examples are:

```
proc F(var x:Int); begin x := - x; end
proc G is GenG(5)
func IsNil(x:DynT)y:boolean; begin y := (x = nil) end
func "+"(a,b:gorp)c:gorp;
    begin
    imports Bias;
    c := gorp'(a.left+b.left+Bias, a.right+b.right+Bias)
    end
```

A routine is a closed scope whose body is a block. Thus its body controls extent for local declarations, but does not inherit identifiers for (non-manifest) objects or labels. The <formals> list declares the identifiers for parameters.

A routine may be a function (func), which returns a value and has no visible side effects; it may be a procedure (proc), which can modify its parameters but must be called as a statement; or it may be a process, which is a potentially-parallel procedure. Special type-specific routines are described in Appendix I.2.

Routine names may be overloaded by binding the same identifier to several definitions with different numbers or types of parameters. The functions for which special infix notation is provided are obvious candidates for overloading.

If a <binding> in a routine header is omitted, it is assumed to be const. The result binding may be used only in procedures. No duplication of identifiers within the <formals> list is permitted, and parameter names may not conflict with declarations or imports in the routine body.

6.4. Exceptions

Some examples are:

```
exception TooBig, TooSmall, Late, Singular
disable TooBig, TooSmall
```

The scope of an exception name is the block in which it is declared. A **disable** declaration in an inner block suppresses detection of the exceptions it names. A handler clause associates recovery code with a statement that may generate an exception (see section 4.7).

The **disable** declaration permits exceptions to be individually suppressed within a given scope. Should an exception occur when its detection is suppressed, the consequences are not defined. An exception must not be signalled or redeclared in a scope in which it is suppressed. Note that suppression of an exception is not an assertion that the condition that gives rise to the exception will not occur.

Standard exceptions will be declared in the global extent.

6.5. Type Definitions

Some examples are:

```
type Counter = Int
type Matrix(n:Int) = array(1..n,1..n) of Real
```

A user may introduce a new type into his program with a type definition. The type definition itself merely introduces the <type name> and defines the representation of the type. Operations are introduced by writing routines whose formal parameters are of the newly-defined type. Scope boundaries, particularly module boundaries, play no role in the definition of the type. There is, as a consequence, no notion of the complete set of operations on a type.

A type definition may be parameterized. The bindings in the formal parameter list must be **const** or **manifest**. If a <binding> is omitted, it will be assumed to be **const**. The names of the formal parameters of the type are available throughout the elaboration of the program as constants, called attributes. They are accessed by treating the <var ident> as a record and the type attribute as a field. Attributes for primitive types are given as part of the type definitions.

Within the scope in which the type is defined, the qualifier **Rep** may be used to indicate that the object named by the identifier **Rep** qualifies is to be treated as if it had the underlying type. This allows operations on the new type to be written using operations on its representation. When no ambiguity arises, the **Rep** qualification may be omitted.

6.6. Generic Definitions

Some examples are:

```
generic proc Reset[T: type](var x:T); begin x := x'min end
proc ResetColor is Reset[Color]
proc ResetX is Reset[Sample]
module Stack is assumed(StackDef)

generic module RingDef[K: Int];
    begin
    exports Ring, Next;
    type Ring = fixed(1,0,0,K-1);
    func Next(R:Ring)N:fixed(1,0,0,K-1); begin N := mod(R+1,K); end
    end
module R5 is RingDef[5]
module R9 is RingDef[9]
```

A generic definition is syntactically like the corresponding specific definition except that it is prefixed by the word **generic** and it may have a set of generic parameters (enclosed in square brackets) after the definition name. For generic definitions, **type** is acceptable as a formal <type name>.

The actual parameters supplied in an instantiation of a generic definition may be any defined identifiers, including those for variables, functions, types, or modules, or any expression. When the generic definition is instantiated, the text of the actual parameters replaces the identifiers that represent the formal parameters. The substitution is done on a lexical, rather than a strictly textual, basis. That is, the identifiers in the generic definition are renamed as necessary to avoid conflicts with the identifiers in the actual parameters.

Both generic definitions and remotely-defined modules or routines may be incorporated in a program as **remote instances**. A remote instance may be an instantiation of a generic definition or a reference to a definition given elsewhere.

A module or routine that is used by the program but whose definition is given elsewhere (e.g., in a library) is incorporated by writing is assumed(<id>) as the body of a module or routine definition. The <id> is used by a pragmat to locate the remote definition.

A generic definition is instantiated by referring to it as the body of a module or routine definition. The effect of the instantiation is as if the generic definition were lexically substituted in place of the reference to it. That is, the body of the module or routine being defined becomes a copy of the generic definition.

An instantiation of a generic definition may be used as the body of a specific module or routine. The usual restrictions on defining new identifiers apply to the module or routine being defined in terms of a generic.

Generic type definitions arise from generic modules. They are instantiated when the module is instantiated. Thereafter, they may be used in declarations or definitions.

If the generic definition has generic parameters, the actual parameters supplied with the instantiation must have correponding types and be syntactically suitable for substitution.

If a generic definition is instantiated more than once in a scope, ambiguous names may be introduced. The usual rules for resolving such ambiguities apply.

6.7. Translation Issues

An example is:

```
prag Optimize(space); Listing(Off) garp
```

A program is a <block>. The extent defined by the outer block of the program is the global extent.

The translator may be guided by <pragmat>s. Pragmats have the same syntax as procedure calls. The set of pragmat names and the interpretations of the arguments are determined by each translator. Translators will ignore pragmats whose names they do not recognize.

A program may be broken into separately defined segments. This decomposition must take place in the global extent. The units of separate definition are modules and routines. The definition

```
module Q is assumed(I)
```

in a segment has the effect of making the semantics of the segment the same as if the (separately defined) text of Q had been substituted for "is assumed(I)". The identifier I refers to a file, library, or other facility for storing separately defined segments. The relation between the identifier I and that storage facility may be established by a pragmat.

It is a matter of optimization whether the separate definition is included as text or separately translated and linked in. In order to perform independent translation of a separately defined component, it is necessary to embed the module or routine being translated in an environment that supplies definitions for all the names it inherits or imports. This environment must form a complete program. It is assumed that the translation system provides commands for selecting which components of such a translation to save and for determining where and in what form they are to be saved.

I. Standard Definitions

I.1. System-Dependent Characteristics

The translator for each system is assumed to provide a module in the global extent that defines appropriate system constants. Such constants are assumed at various points in the language definition; these and certain others are summarized here in the form of a skeleton module.

```
module Sys;
begin
exports . . .                ! exports all definitions below

type Int = fixed(. . .)      ! appropriate to the machine
                             ! Note Int.Min and Int.Max give range

type Real = float(. . .)     ! appropriate to the machine
                             ! Attributes give range, precision, scale

const . . .                  ! constants that descibe properties of the
                             ! object machine

proc . . .                   ! procedures for accessing facilities of the
                             ! operating and file systems

exceptions . . .             ! System-defined exceptions such as Assertion, BadAssign....

end
```

I.2. Properties of Types

All types have assignment operators and routines for conversion to appropriate other types. In particular, the scalar types have routines for converting to and from character strings. All nonscalar types have constructors. The sections below sketch some important properties of the built-in types.

I.2.1. Fixed

Literals:	digit strings
Attributes:	Min, Max, Precision, Scale
Infix operations:	Arithmetic and relational
Special routines:	rounding, truncation

I.2.2. Float

Literals:	digit strings with decimal point
Attributes:	Min, Max, Radix, Precison, MinExp, MaxExp
Infix operations:	Arithmetic and relational
Special routines:	rounding, truncation

I.2.3. Enumerations

All enumerations are ordered. The literals are assumed to appear in the declaration in increasing order.

Literals:	As given in definition
Attributes:	Min, Max
Infix operations:	Relational
Special routines:	succ, pred

I.2.4. Boolean

Literals:	true, false
Attributes:	none
Infix operations:	logical
Special routines:	none

I.2.5. Characters

Literals:	Quoted characters
Attributes:	Min, Max
Infix operations:	none
Special routines:	as for enumerations

I.2.6. Latches

A latch is a simple spinlock for mutual exclusion. If the latch is open, it is available for siezure; if it is closed, a Lock command will wait on it.

Literals:	open, closed
Attributes:	none
Infix operations:	none
Special routines:	Lock, IfLock, Unlock

I.2.7. Arrays

Literals:	none
Attributes:	Range, EltType
Infix operations:	none
Special operations:	subscript, subarray, catenation, upper bound, lower bound

I.2.8. Sets

"Sets" are boolean vectors on which some additional operations are defined.

Literals:	empty
Attributes:	EltType, MaxSize
Infix operations:	logical
Special operations:	subscript

I.2.9. Dynamic Types

Literals:	nil
Attributes:	The named variable does not itself have attributes, but the dynamic variable that it references may.
Infix operations:	none
Special operations:	.all denotes whole value of dynamic object, as distinguished from the reference. A dynamic constructor allocates a new dynamic object.
Special routines:	none

I.2.10. Records

Literals:	none
Attributes:	individually defined with record type
Infix operations:	none
Special operations:	field selection, constructors
Special routines:	none

I.2.11. Variants

Literals:	none
Attributes:	individually defined with variant type
Infix operations:	none
Special operations:	variant must be designated to reference contents
Special routines:	none

I.2.12. Strings

Literals:	Quoted strings
Attributes:	Length
Infix operations:	none
Special operations:	subscript, substring, catenation

I.2.13. Activations

Literals:	mint
Attributes:	none
Infix operations:	none
Special operations:	create
Special routines:	To change state: Activate(A), Suspend(A), UnlockAndSuspend(A,L), UnlockAndActivate(A,L), LockAndSuspend(A,L), LockAndActivate(A,L), Terminate(A) To query state: IsMint(A), IsAct(A), IsSusp(A), IsTerm(A) To obtain actname: NameOf(A), Me() To sent exception: Notify(A) Other: Priority(A), SetPriority(A), Time(A)

where A is an activation or actname and L is a latch

Assignment causes the BadAssign exception if either the value or the variable to which it is being assigned is in a state other than mint.

I.2.14. Actnames

Literals:	mint
Attributes:	none
Infix operations:	none
Special operations:	none
Special routines:	Same as for activations

I.2.15. Files

A minimal input-output facility will be provided.

I.3. Alphabets

The following context-free substitutions reduce the alphabet used in this report to the standard 64-character ASCII subset. Note that some identifiers are pre-empted as a result.

For the publication character:	Substitute the ASCII string:
lower case a..z	upper case A..Z
≤	<=
≥	>=
≠	<>
∧	and
∨	or
{	<<
}	>>

II. Collected Syntax

```
<const>       ::= <digit>+ { . <digit>+ }* | true | false | nil | closed | open | mint | empty
                | <constructor> | <id> | <qual id> ' <const> | <type> ' <const> | <expr>
<constructor> ::= ( <expr>,* ) | ( { <option> -> <expr> },+ ) | " <char>* "
<var id>      ::= <qual id> | <var id> ( <actuals> ) | <var id> . <id> | <var id> ( <range> ) | Rep' <id>
<range>       ::= <expr> . . <expr> | <type>
<option>      ::= { <const> | <range> },+
<qual id>     ::= { <id> '}* <id>
<id>          ::= <letter> <letter or _ or digit>*
<expr>        ::= <unop>* <var id> | <unop>* <const> | <unop>* <func call>
                | <unop>* ( <expr> ) | ( <expr> ) . <id> | <expr> <binop> <expr>
<unop>        ::= - | ¬
<binop>       ::= * | / | + | - | < | ≤ | > | ≥ | = | ≠ | ∧ | cand | ∨ | cor | ↑
<func call>   ::= <qual id> ( <actuals> )
<actuals>     ::= <expr>,*
<stmt>        ::= <proc call> | <id> : <stmt> | <empty> | <block>
                | <var id> := <expr>
                | if <expr> then <stmt>;* { elif <expr> then <stmt>;* }* { else <stmt>;* }* fi
                | case <expr> { on <option> -> <stmt>;* }+ esac
                | while <expr> do <stmt>;* od | for <id> in <range> do <stmt>;* od
                | goto <id>
                | signal <qual id> | resignal | assert <expr>
                | <stmt> { { on <id>,+ -> <stmt>;* }+ }
                | create <var id> ( <actuals> )
<proc call>   ::= <qual id> ( <actuals> )
<block>       ::= <code body>
<code body>   ::= begin { <def-decl> ; }* <stmt>;* end
<type>        ::= fixed( <actuals> ) | float( <actuals> ) | boolean | latch | char | file( <actuals> )
                | enum( <id>,+ ] | enum( { " <char> " },+ ] | <expr> . . <expr>
                | set( <actuals> ) | string( <actuals> )
                | array ( <range>,+ ) of <type> | record [ <declaration>,+ ]
                | variant <declaration> [ { on <option> -> <type> }+]
                | dynamic <type> | activation of <qual id> | actname
                | <type name> { ( <actuals> ) }*
<type name>   ::= fixed | float | boolean | latch | char | file | set | string
                | enum( <id>,+ ] | enum( { " <char> " },+ ]
                | array [ <type name>,* ] of <type name> | record [ { <id>,+ : <type name> },+ ]
                | variant [ <type name> { on <option> -> <type name> }* ]
                | dynamic <type name> | activation [ <qual id> ] | actname
                | <qual id> { [ <qual id>,+ ] }*
<def-decl>    ::= <declaration> | <mod def> | <routine def> | <type def> | <generic def> | <empty>
                | imports <qual id>,+ | exports <qual id>,+ | exception <id>,+ | disable <id>,+
                | prag <proc call>;+ ;* garp
<declaration> ::= <binding> { <id>,+ { : <type> }* { := <expr> }* },+ | <binding> { <id>,+ : <type name> },+
<mod def>     ::= module <id> <mod text>
<mod text>    ::= ; <code body> | <remote inst>
<routine def> ::= proc <id> <proc text> | func <id> <func text> | process <id> <proc text>
                | func " { <unop> | <binop> } " <func text>
<func text>   ::= ( <formals> ) <id> : <type> ; <block> | <remote inst>
<proc text>   ::= ( <formals> ); <block> | <remote inst>
<type def>    ::= type <type name> { ( <formals> ) }* = <type>
<generic def> ::= generic module <id> [ <formals> ] <mod text> | generic func <id> [ <formals> ] <func text>
                | generic proc <id> [ <formals> ] <proc text> | generic process <id> [ <formals> ] <proc text>
<remote inst> ::= is <qual id> [ <actuals> ] | is assumed ( <id> )
<formals>     ::= { <binding> <id>,+ : <type name> },*
<binding>     ::= <empty> | var | const | manifest | result
```

ANHANG 7

CMU-CS-78-132

TARTAN

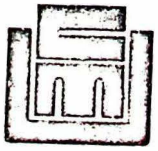
Language Design for the Ironman Requirement: Notes and Examples

Mary Shaw
Paul Hilfinger
Wm. A. Wulf

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

June, 1978

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

CMU-CS-78-132

TARTAN

Language Design for the Ironman Requirement: Notes and Examples

Mary Shaw
Paul Hilfinger
Wm. A. Wulf

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

June, 1978

Abstract: The Tartan language was designed as an experiment to see whether the Ironman requirement for a common high-order programming language could be satisfied by an extremely simple language. The result, Tartan substantially meets the Ironman requirement. We believe it is substantially simpler than the four designs that were done in the first phase of the COD-1 effort. The language definition appears in a companion report; this report provides a more expository discussion of some of the language's features, some examples of its use, and a discussion of some facilities that could enhance the basic design at relatively little cost.

This work was supported by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074 (monitored by the Air Force Office of Scientific Research).

Tartan: Notes and Examples

1. Notes on Important Issues 1

1.1. Vocabulary 1
1.2. Scope and Extent 2
1.2.1. Scope 2
1.2.2. Extent 2
1.3. Modules and Routines 3
1.3.1. Modules 3
1.3.2. Routines 3
1.4. Generic Definitions 4
1.4.1. Writing and Using Generic Definitions 4
1.4.2. Separate Definitions 5
1.5. Types 6
1.5.1. Characteristics and Attributes 7
1.5.2. Type Names 7
1.5.3. Array Types 8
1.5.4. Declarations 8
1.5.5. Type Checking 8
1.5.6. Defining Types 9
1.5.7. Operations on New Types 9
1.6. Parallel Processes 9
1.6.1. Activations 10
1.6.2. Fork and Join 10
1.6.3. Activation Names 11
1.7. Unresolved Issues 11
1.7.1. Machine-Dependent Code 11
1.7.2. Simulation 11
1.7.3. Definition of Integers 12
1.7.4. Low-Level Input and Output 12
1.7.5. Higher-Level Synchronization 12

2. Programming Examples 13

2.1. Simple Static Data Type 13
2.2. Simple Dynamic Data Type 13
2.3. Selecting Representations 14
2.4. Safe Data 15

3. Optional Additions to the Language 16
References 16

1. Notes on Important Issues

The Tartan reference manual is the defining document for the Tartan language. However, some of the facilities designed in response to the Ironman requirement deserve more unified and expository explanations than can be included in a reference manual. This chapter discusses the Tartan solutions to several important problems posed by the Ironman requirement.

The Tartan language draws heavily on the Pascal tradition. Both the reference manual and these notes assume familiarity of Pascal-like languages. These notes also assume familiarity with the Ironman requirements [1] and the Tartan reference manual [2].

1.1. Vocabulary

A Tartan program is made up of definitions, declarations, and (executable) statements. A definition binds an identifier to a module, routine (function, procedure, or process), type, or exception: it is processed during translation. A declaration binds an identifier to an object (i.e., a variable or value); it is processed at run time, usually to allocate storage. Executable statements are elaborated at run time to effect computations -- manipulation of values.

Identifiers can be bound to modules, routines, types, objects, statements, and exceptions. Individual identifiers are qualified with the names of the modules in which they are defined in order to avoid conflicts with names declared in other modules.

The computation described by a program is carried out by **elaborating** the program. We use the word "elaboration", in preference to "execution", to connote actions taken during translation as well as those taken during execution. Elaboration may be thought of as an idealized direct execution of the textual version of the program. The effect of elaborating each construct in the language is given in the reference manual.

Although the language prohibits making a declaration that gives new meaning to an identifier in a given scope, duplicate identifiers might arise in three situations. These situations, and the way Tartan deals with them, are:

- The same identifier is exported from two modules. The ambiguity is prevented by name qualification: All identifiers exported from a module are prefixed with the name of the module that exported them; the prefix is separated from the identifier by an apostrophe. Thus if identifier x is exported to the same scope by both modules M and N, we write

```
M'x                    ! for the x exported from M
N'x                    ! for the x exported from N
```

  The qualification may be omitted if no ambiguity arises.

- An identifier is used as an overloaded routine or type name. That is, the same routine name is given several definitions with different numbers or types of parameters. Operator overloading is permitted so that similar operations on distinct types, particularly infix operations, can be given the same names. The identifiers for the routines or types are disambiguated by examining the parameter types and choosing the routine whose formal parameter types are matched by the types of the actuals. A similar situation exists with identifiers for families of related types. In order to discuss these situations, we introduce the notion of **signature**:

  - The signature of a routine is the routine name together with its formal parameter types. The type of the value returned by a function is not included in its signature.

  - The signature of a type is its simple type name together with its generic characteristics. Generic characteristics are discussed in Section 1.5.1.

- A literal or constructor might potentially be of two or more different types. The ambiguity is resolved by qualifying the literal or constructor with the intended type, including its attributes.

1.2. Scope and Extent

Scope determines the interpretation of identifiers, so all the text in a given lexical scope shares the same vocabulary -- definitions, variables, etc. **Scope rules** permit some identifiers to be used with the same interpretation in multiple lexical scopes.

The **extent** of a variable is its lifetime -- the time during which it affects or is affected by the elaboration of the program. The interaction of control and lexical structure determines extent. Binding is the association of identifiers with program entities (objects, modules, routines, types, statements, and exceptions). The bindings in effect at any time result from the interaction of control and lexical scope.

1.2.1. Scope

Lexical structure is imposed on Tartan programs by blocks and modules, which delimit lexical scopes. There are no restrictions on the ways these scopes may be nested. Both constructs may use identifiers defined in other scopes; both may define identifiers that can be used in other scopes. Scope rules govern the legal bindings of identifiers in a lexical scope to program entities; they also control the importing and exporting of identifiers to other scopes. Blocks and modules differ only in their scope rules and in their effects on the extent of variables. Tartan has two scope rules:

- An **open** scope inherits (imports automatically) all the identifiers that are defined in its enclosing scope. It may not export any identifiers to its enclosing scope. Blocks are open scopes except when used as routine bodies.

- A **closed** scope inherits all identifiers that are defined in its enclosing scope except those for labels and objects.[1] It may explicitly import identifiers for objects (variables and constants), provided they have global extent. A closed scope that is a module may export identifiers that name variables, definitions, or exceptions; the exported identifiers have the status of any other identifiers defined in the enclosing scope. All modules are closed scopes, as are blocks when they are used as routine bodies.

Identifiers that are exported from an inner scope or imported from an outer scope have the status of identifiers defined in the scope. Redefinition of identifiers within a scope is not permitted. The convenience of being able to do so does not offset the danger of confusion. This does not, however, prohibit overloading of routines names; the differences in signatures suffice to prevent confusion. In addition, the same identifier might be imported with different meanings from two different modules; such identifiers are qualified with the names of the modules in which they were defined. Thus they are not duplicate definitions. Similarly, literals and constructors are qualified with their types to prevent ambiguity. In either case, the module or type qualifier may be omitted if no ambiguity arises.

1.2.2. Extent

Extent rules govern the lifetimes of objects. Extent is controlled by blocks, independent of whether they correspond to open or closed scopes. Nothing except blocks controls extent. The static data of a block is allocated when the declarations of the block are elaborated (in lexical order) at block entry. It is deallocated when the block is exited or terminated. Note that modules do not define extents, so the extent of data defined in a module coincides with the extent of its surrounding block.

Values of dynamic types point to dynamically allocated variables. The type of object that may be pointed at is part of a dynamic type. The extent of dynamically allocated variables is coincident with the blocks in which the associated dynamic types are declared. Since type names are not accessible outside the blocks in which they are defined, no references can outlive the block with which the extent is associated.

---

[1] Literals and identifiers for variables that are declared manifest are inherited.

1.3. Modules and Routines

Modules and routines are closed scopes. Modules serve as an encapsulation mechanism, protecting the privacy of definitions and declarations without restricting their extent. Routines are used for program structuring and abstraction of operators; they define operations that may be invoked during elaboration of a program.

1.3.1. Modules

A module is a closed scope that allows local definitions to be shared without making them public. It also serves to bundle up related definitions for administrative (program organization) purposes. It may export identifiers for definitions and objects to the scope in which it is defined. A module has no parameters.

A module is purely a scope-defining device. Its elaboration takes place during the elaboration of declarations for the block in which the module is defined. This elaboration consists of elaborating the definitions and declarations of the module in lexical order, then elaborating the statements of the module.

A module or routine inherits identifiers for definitions (modules, routines, types, and exceptions), literals, and manifest objects from its enclosing scope. It may explicitly import identifiers of objects from that scope, provided the objects have global extent. A module, but not a routine, may export identifiers other than labels to its enclosing scope.

1.3.2. Routines

A routine is a closed scope whose body is a block. Thus its body controls extent for local declarations, but does not inherit identifiers for variables or non-manifest constants. The <formals> list declares the identifiers for parameters.

A routine may be a function (func), which returns a value and has no visible side effects; it may be a procedure (proc), which can modify its parameters but must be called as a statement; or it may be a process, which is a potentially-parallel procedure. Special type-specific routines for many types are listed in the Tartan Reference Manual.

The symbols for the unary and binary operators are used as routine names in order to provide overloaded definitions for those operations.

If a <binding> in a routine header is omitted, it is assumed to be const. The result binding may be used only in <formals> lists of procedures. Functions are permitted to specify var parameters in order to avoid the copy associated with const.[1] However, as noted below, visible side effects on such parameters are prohibited. No duplication of identifiers within the <formals> list is permitted. Further, formal parameter names may not conflict with declarations or imports in the routine body.

If a routine name is overloaded, the definition whose signature matches the call is selected.

During elaboration of a function, assignment to a variable that is not local to the function body (or to the body of any routine it invokes, directly or indirectly) is permitted only if the function is never invoked in a scope where such a change is made to a variable or component that is directly accessible by the caller. Such variables may be imported by the function from a module within which the function is defined. They may also be fields of var parameters if the type of the parameter is defined in the same module as the function and the field name is not exported. An example of the latter case appears in section 2.4.

This is a compromise solution to the side-effect problem. Many routines are quite reasonably coded as value-returning: Get of section 2.4, monitoring routines, random number generators, and Pop for stacks. However, the current state of the art does not offer a sharp rule from distinguishing safe from unsafe side effects.

---

[1] In the presence of parallelism, it may not be safe to optimize away the copy of a const parameter even if the routine does not alter it.

Actual parameters are matched with formal parameters positionally. They must satisfy restrictions on type, binding and aliasing.

- The type of an actual parameter is acceptable if its <type name> exactly matches the <type name> of the corresponding formal parameter. Type attributes (instantiation parameters of a type) play no role in type checking.

- The binding of the actual parameter is acceptable if it matches the <binding> of the corresponding formal parameter according to the following rules:

If the formal parameter is	then the actual parameter may be
var	<var id>
const	<expr>
manifest	any manifest <expr>
result	<var id>

- Finally, the set of actual parameters must satisfy the following nonaliasing restriction: A variable may not be used in more than one var or result position of a single procedure or process call. For the purpose of testing this restriction, imported variables are considered to be var parameters.

1.4. Generic Definitions

A facility for making generic definitions is provided in order to allow the programmer to write a single textual definition that serves as an abbreviation for many closely-related specific definitions. Modules and routines may be defined generically.

A generic definition is instantiated by referring to it as the body of a module or routine definition. The effect of the instantiation is as if the generic definition were lexically substituted in place of the reference to it. That is, the body of the module or routine being defined becomes a copy of the generic definition.

1.4.1. Writing and Using Generic Definitions

A generic definition is syntactically like the corresponding specific definition except that it is prefixed by the word generic and it may have a set of generic parameters (enclosed in square brackets) after the name of the construct being defined. The parameters may be any defined identifiers, including those for variables, routines, types, or modules, or any expression. When the generic definition is instantiated, the text of the actual parameters replaces the identifiers that represent the formal parameters. The substitution is done on a lexical, rather that a strictly textual, basis. That is, the identifiers in the generic definition are renamed as necessary to avoid conflicts with the identifiers in the actual parameters.

For example, the collection of functions

```
func F2(X:Int)y:Int; begin y := 2 * X end
func F3(X:Int)y:Int; begin y := 3 * X end
func F4(X:Int)y:Int; begin y := 4 * X end
   and so on
```

can be defined by the generic definition

```
generic func F[Mult:Int] (X:Int)y:Int; begin y := Mult * X end
```

and the specific instantiations

```
func F2 is F[2]
func F3 is F[3]
func F4 is F[4]
   and so on
```

An instantiation of a generic definition may be used as the body of a specific module or routine. The usual restrictions on defining new identifiers apply to the module or routine being defined in terms of a generic.

Generic type definitions arise from generic modules. They are instantiated when the module is instantiated. Thereafter, they may be used in declarations or definitions.

If the generic definition has generic parameters, the actual parameters supplied with the instantiation must have correponding types and be syntactically suitable for substitution.

If a generic definition is instantiated more than once in a scope, ambiguous names may be introduced. The usual rules for resolving such ambiguities apply.

1.4.2. Separate Definitions

Tartan supports separate definitions, and potentially separate compilation, in the same way as it supports generic definitions. A program may be broken into separately defined segments. This decomposition must take place in the global extent. The units of separate definition are modules and routines. The definition

```
module Q is assumed(I)
```

in a segment has the effect of making the semantics of the segment the same as if the (separately defined) text of Q had been substituted for "is assumed(I)". The identifier I refers to a file, library, or other facility for storing separately defined segments. The relation between the identifier I and that storage facility may be established by a pragmat.

Suppose we want to develop and maintain a program with the following structure:

```
begin
module COM; begin export X; . . . end;
module M1; begin import X,Y; export Z; . . . end;
module M2;
    begin import X,Z;
    export W;
    module M3; begin . . . end;
     . . .
    end;
var Y: . . .;
! Main program using W, X, Y, Z
end;
```

If the definitions of COM, M1, and M2 are stored in a library, the following program will have the same effect:

```
begin
prag Require(ComDef,LIB.COM.TXT); Require(M1Def,LIB.M1.REL);
   Require(M2Def,LIB.M2.REL); garp;
module COM is assumed (COMDef);
module M1 is assumed (M1Def);
module M2 is assumed (M2Def);
var Y: . . .;
! Main program using W, X, Y, Z
end;
```

We assume here that the second argument of the Require pragmat is interpreted by the system as a pointer into a library. From the standpoint of the language, it is a matter of optimization whether the separate definitions are included as text or separately translated and linked in.

In order to perform independent translations of a separately defined module, it is necessary to embed it in an environment that provides the definitions it depends on. This environment must form a complete program. The translation system is assumed to provide commands for selecting which components of such a translation to save and for determining where and in what form they are saved. In the examples here, we will simulate that facility with a pragmat located outside the program. In the example above, module COM does not depend on any external definitions. In order to compile it separately, we write simply:

```
prag Save(Com,LIB.COM.TXT); garp;
begin
module COM; begin export X; . . . end;
end
```

Module M1 depends on the X exported from COM and the Y declared in the main program. To translate M1 separately, we must therefore write:

```
prag Save(M1,LIB.M1.REL); garp;
begin
prag Require(ComDef,LIB.COM.TXT); garp;
module COM is assumed (COMDef);
module M1; begin import X,Y; export Z; . . . end;
var Y: . . .;
end
```

If module M2 were translated monolithically, its translation environment would look much the same. Suppose, however, that the definitions of M2 and M3 are to be separated. They can be translated independently with the following two programs:

```
prag Save(M2,LIB.M2.REL); garp;
begin
prag Require(ComDef,LIB.COM.TXT); Require(M1Def,LIB.M1.REL);
   Require(M3Def,LIB.M3.REL); garp;
module COM is assumed (COMDef);
module M1 is assumed (M1Def);
module M2;
    begin import X,Z;
    export W;
    module M3 is assumed(M3Def);
     . . .
    end;
end
```

```
prag Save(M3,LIB.M3.REL); garp;
begin
prag Require(ComDef,LIB.COM.TXT); Require(M1Def,LIB.M1.REL); garp
module COM is assumed (COMDef);
module M1 is assumed (M1Def);
module M2;
    begin import X,Z;
    export W;
    ! Only the declarations of M2 that are required by M3 appear
    module M3; begin . . . end;
    end;
end
```

1.5. Types

The notion of type is introduced into languages to govern the ways operations are applied to objects. Types determine certain properties of data (values), including what operations on the values are legal and precisely what their effects are. Every object has a fixed type. This type is determinable during translation. The <type name> is determined by the signature of the type as described in section 1.5.2. Tartan provides certain built-in types; these include both simple and composite types. The user may define new types on the basis of these primitives. Both user-defined and built-in types are used to ensure that the actual parameters passed to a routine match the corresponding formal parameters. The types of the formal parameters are also used to construct the signature of a routine in order to resolve overloading ambiguities.

In Tartan, every value has exactly one type. This type is determined

- by the declaration of a variable or definition of a function
- by the lexical form and context of occurrence of a literal

Types appear in four contexts:

- in declarations, to give the type and attributes of an object
- in type definitions, to give the base representation of a newly-defined type
- in formal parameter lists, to restrict the objects that may be passed as parameters
- in function definitions, to give the type of the result

1.5.1. Characteristics and Attributes

Some of the properties of a type are the same for all values and objects of the type. These are called generic characteristics and are discussed below. Other properties of a type, called attributes, may differ from one value or object of the type to another. For example, in Tartan the type of the values used to index the elements of an array (the type of the index set) is a generic characteristic, whereas the exact bounds of the array (which values are in the index set) are attributes.

The set of attribute names associated with a type and the types of the corresponding attribute values are given in the definition of the type. For example, objects of type fixed have attributes Max, Min, Precision, and Scale.

Note that the attributes values of an object are not part of its type. It is therefore possible to write routines that operate on objects with different attributes. For example, it is straightforward to write routines that operate on arrays of arbitrary size.

It is often convenient to define families of related types with similar properties, and in which the differences can be captured through differences in generic properties. A type definition parameterized in this way can be cast as a generic type definition. Members of the family with distinct characteristics are distinct types.

Generic types are introduced through generic module definitions. For example,

```
generic module Blocker[T:type];
    begin
    type Block[T](Order:Int) = array(1..Order) of T;
    proc BlockIt(var B:Block[T]); begin . . . end
    end
```

defines a set of types Block[...] and a set of corresponding procedures. The definitions

```
module IntBlock is Blocker[Int];
module RealBlock is Blocker[Real];
module MyBlock is Blocker[MyType];
```

introduce, respectively, the types

```
Block[Int](Order:Int)
Block[Real](Order:Int)
Block[MyType](Order:Int)
```

each of which has an Order attribute. Note also that the procedure BlockIt is overloaded to operate on all these types, and that it is indifferent to the Order attribute of its argument.

1.5.2. Type Names

In Tartan, a <type name> may be either a simple identifier or an identifier inflected with additional type names. The <type name> so formed captures the signature of the type. For example, the <type name>s in the example above are

```
Block[Int]
Block[Real]
Block[MyType]
```

Although the definitions of these three types are closely related (they arise from instantiations of the same generic module), the types are entirely distinct.

The <type name>s for the primitive scalar and simple nonscalar types are the keywords used to declare them: fixed, float, boolean, latch, char, set, string, actname, file.

The <type name> for an array declared "array(a..b) of D" is "array[I,D]", where I is the <type name> of a and b. See section 1.5.3 for the derivation.

The <type name> for an enumeration declared enum[L1,L2,...Ln] is enum[L1,L2,...,Ln].

The <type name> for an activation declared activation of P is activation[P].

The <type name> for a dynamic type declared dynamic T is dynamic T.

The <type name> for a record type is based on the sequence of field names and <type name>s in its declaration. For a record declared "record[F1:T1, F2:T2, ..., Fn:Tn]" the <type name> is "record[F1:TN1, F2TN2, ..., Fn:TNn]", where the Fi are lists of field names, the Ti are types, and the TNi are type names. Bindings in the declaration do not appear in the type name. Thus, in the code

fragment

```
proc P(var x:record[a,b:Real]); begin . . . end;
var y:record[a,b:Real];
var z:record[c,d:Real];
```

variables y and z have different <type name>s and only y is acceptable as a parameter to P.

The <type name> for a variant is "variant[TT,T1->V1,T2->V2,...,Tn->Vn]", where TT is the <type name> of the tag, Ti is the i^{th} value of the tag type, and Vi is the <type name> that corresponds to the i^{th} value of the tag type. As a result, two variant <type>s are the same if they specify the same <type>s for all values of the tag. Thus for

```
type Color = enum [red, green, blue, yellow];
variant T:Color [on red -> x:Int on blue -> y:Mark(5) on others -> z:array(1..5) of Int]
```

the <type name> is "variant[Color, red->Int, green->array[Int,Int], blue->Mark, yellow->array[Int,Int]]".

The <type name> for a defined type is the type name given in the type definition, as illustrated above for Block[...].

1.5.3. Array Types

The built-in array type is in fact a generic family. Arrays have uniform properties in that every array is a structure for storing a linear homogeneous fixed-length sequence of values indexed by a given ordered set of values. However, arrays with different element types or different types of indices are distinct types.

This particular generic family of types is so common that Tartan, like most languages, provides special syntax for it. There is a set of types pre-defined as "array[IxType,EltType](r)" where IxType is the index type, EltType is the element type, and r is a (sub)range of IxType. The syntax "array(r) of EltType" is provided as an abbreviation for each such type. Thus "array(1..10) of float" means "array[int,float](1..10)". Its type name, "array[int,float]", is written "array[int] of float". Thus if we have declared

```
var V: array (1..10) of Float
var B: array (red..green) of boolean
```

the generic type of both B and V is array, but their <type name>s are different. The <type name> of B is array[int,float], whereas the <type name> of V is array[color,boolean].

The type "array(A,B) of T" is an abbreviation for "array(A) of array(B) of T". Similarly, the array accessor "V(i,j)" is an abbreviation for "V(i)(j)".

1.5.4. Declarations

The attributes of a variable become fixed at the time of its allocation. For static variables, this occurs during elaboration of the declaration. Variables of dynamic types do not themselves have attributes. The dynamically allocated objects they refer to do, however, have attributes; these are supplied whenever a constructor is executed.

The declaration of a static variable must provide both a <type name> and values for the attributes associated with that type. For example, the declaration "var V: array (m..n) of Int", which is an abbreviation for "var V: array[Int,Int](m..n)", computes the current values of m and n to obtain the range of the index set, then statically allocates a suitable block of storage. However, the program fragment

```
type Arr(n:Int) = dynamic array (1..n) of Int;
var V: Arr;
V := Arr(5)();
```

allocates the variable V with type Arr, no attributes, and all values undefined. The declaration allocates a reference to V and sets it to nil. The constructor dynamically creates a new object of type array(Int) of Int with subscript range attribute "1..5" and associates this object with variable V. A subsequent assignment to V might use a constructor with a different bound.

1.5.5. Type Checking

The type checking rule for matching actual and formal parameters is based on the types (but not the attributes) of the parameters. The actual parameter is acceptable iff the <type name> from its declaration exactly matches the <type name> of the formal parameter.

The attributes of the values returned by a function invocation are determined immediately before calling the function. They must therefore be specified in terms of input values of the function. For example, if Str is a type with attribute Length, the definition

```
func Concat(S, T: Str)R:Str; begin . . . end;
```

would not be legal, since the attributes of the functional result are not specified. The following, however, would both be legal (but would have different meanings):

```
func Concat(S,T: Str)R:Str(27); begin . . . end;
func Concat(S,T: Str)R:Str(S.Length+T.Length); begin . . . end;
```

This simplifies the implementation, but it precludes the definition of functions that return values whose attributes can only be determined during the evaluation of the function. This should not usually be a stringent constraint; in the worst case a dynamic type may be used to return the value.

1.5.6. Defining Types

A user may introduce a new type into his program with a type definition. The type definition itself merely introduces the <type name> and defines the representation of the type. Operations are introduced by writing routines whose formal parameters are of the newly-defined type. Scope boundaries, particularly module boundaries, play no role in the definition of the type. There is, as a consequence, no notion of the complete set of operations on a type.

A type definition may be parameterized with attributes. The bindings in the formal parameter list must be const or manifest. If a <binding> is omitted, it will be assumed to be const. The names of the formal parameters of the type are available throughout the elaboration of the program as constants, called attributes. They are accessed by treating the <var ident> as a record and the type attribute as a const field. Attributes for primitive types are given as part of the type definitions.

1.5.7. Operations on New Types

Operations on new types are introduced by routine definitions. These may be either routines called with normal invocation syntax or definitions for infix functions. In order to make it possible to write basic operations on the new type, Tartan provides a means of applying operations of the underlying representation to objects of the new type. Within the scope in which the type is defined, the qualifier Rep may be used to indicate that the object named by the identifier it qualifies is to be treated as if it had the underlying type. It is not exportable. This allows operations on the new type to be written using operations on its representation. When no ambiguity arises, the Rep qualification may be omitted. For example, we may write

```
type Mark = Int;
func "+"(a,b: Mark)c:Mark; begin Rep'c := Rep'a + Rep'b end;
```

Rep qualification is intended to be used within a module in order to write primitive operations and to extend operators to the new type. It is obviously possible to abuse the facility.

An assignment operator is automatically supplied for user-defined types. Although it may be invoked with any variable and value of the type, it signals the BadAssign exception if the attributes of its left and right operands are not identical or if component-by-component assignment would fail. Sizes of nonscalars are thus guaranteed to be compatible. Clearly, assignment may be well-defined in cases where this rule disallows it. Such assignment operators could be provided if user-defined assignment were compatible with the requirements.

When a module is used to encapsulate the definition of a type and its operations, the type name and some of the operations must be exported from the module. Types, named routines, field accessors for records, and variables are exported by including their names in the exports list of the module. The right to apply infix operators, constructors, subscripts, ".all", or the create command are exported by including the special names T'infix, T'constr, T'subscr, T'all, and T'create, respectively, in the exports list. Literals of enumerated types are exported automatically if the types are exported.

1.6. Parallel Processes

Parallel processes are controlled with data of two types -- activations of processes and actnames, or names of activations. An activation variable must be an instantiation of a given process; it may contain at most one activation of that process during its lifetime. An actname variable is a pointer to an activation. A single actname may be associated with different instantiations of different processes

from time to time.

Processes are similar to procedures. The syntactic distinction between procedures and processes is imposed because we believe the potential for parallel execution should be indicated explicitly in the program.

Note that activations and actnames control only the parallel control flow of the program. No synchronization is supplied with the processes; this must be coded explicitly with the primitive latches or with other, nonprimitive synchronization.

1.6.1. Activations

Activations of processes are used to control parallel or pseudo-parallel execution of instances of the named process. If P is a process and x is a variable of type activation of P, then x can contain an independently-executing instantiation of P, called an activation of P. An activation of P may be in one of several states:

- Mint: A mint activation has not yet been started up as a process. The only operations that can be performed on it are create, NameOf (i.e., the function that returns the activation's name), and the state-interrogation predicates. A newly-declared activation or actname is initialized to the literal mint.

- Suspended: A suspended activation can have no effect on any objects; in essence, it is not executing and will not execute until it is activated (see below).

- Active: An active activation is one in which it is feasible for elaboration to take place. It may affect objects, and its clock may advance.

- Dead: A dead activation admits of no further elaboration. It cannot be revived and it can play no further role in the program. An activation becomes dead when it exits normally, when it fails to handle an exception raised during its elaboration, or when it is named by a Terminate command.

The extent of an activation variable is determined by the block in which it is declared. When such a variable is declared, an activation of the named process is instantiated, set to state mint, and associated with the declared process name. The immediately enclosing block cannot be exited until all activations declared within it are dead or still mint. An activation is associated with exactly one process, but a single process may be instantiated multiple times for different activations.

If x has been declared as an activation of P and is in mint state, the statement "create x(...)" creates a new activation of P in suspended state. The formals of P are bound to the actuals supplied in the create in the same way as actuals are bound for a procedure call. If a process takes a var parameter, the corresponding actual parameter must have extent at least as great as the activation's extent. For purposes of this rule, an activation passed as a var parameter to a routine is treated as if its scope were that of the process definition. As a result, translators need no dynamic extent checking.

Except for create, all operations on activations are syntactically routine invocations. These routines conrol the processes and hence the parallelism by changing and interrogating the state of individual activations. They are listed in the Tartan Reference Manual.

1.6.2. Fork and Join

The extent rules require each activation to complete (exit or terminate) or still be mint before the block in which it is declared can exit. This provides an implicit join operation. A fork can be obtained with a series of creates and activates. For example,

```
begin
process P(const x:Int); begin . . . end;
var V: array(1..10) of activation of P;
for i in 1..10 do create V[i](i); activate(P[i]) od
. . .
end
```

declares ten activations of a process, uses create to start them up with different values of the input

variable (using the loop index as the input value as well as to index the array of activations), moves each activation into active state, and waits at the end of the block for the activations to terminate. After starting the activations of P, the main program may continue with other computation, monitor the progress of the activations, or simply wait for the activations to terminate.

1.6.3. Activation Names

An actname may name any activation. An actname variable is not permanently associated with any particular activation, and there is no requirement about the state of the activation named by an actname when the extent of that actname variable is exited or terminated. This permits routines to operate on activations without knowing what processes they are activations of. For example, it makes it possible for routines that are generally useful for managing activations to be defined in a large scope without requiring all process definitions and activation variables to include that scope. A single activation may be named by more than one actname. There is no dangling reference problem: Even though the reference (actname) may outlive the activation, the activation will be dead (terminated or mint) after its block is exited (and thus no unexpected computational results can be induced).[1] Since the create command cannot be applied to an actname, the process cannot be restarted.

Activation variables may not be the objects of assignments and may not appear in result parameter positions. However, each activation has a name, of type actname. This name may be obtained by invoking the function NameOf on an activation. All operations on activations except create extend to actnames. Thus, Suspend(NameOf(x)) has the same effect as Suspend(x) The special operation Me() returns the actname of the current process. In addition, actname variables may appear in assignments. (Thus users may write programs that operate on anonymous activations, for example to do special-purpose scheduling.) The extent of an actname variable may dominate the extent of the activation it names. If that situation arises, after the extent of the activation is exited, the actname will refer to a terminated process, and no damage can be done.

The Notify operation on activations or actnames signals the Terminate exception in the currently-executing statement of the activation named by the command. Within the activation in which it is raised, Terminate is treated like any other exception. This is the only mechanism provided by Tartan that enables one activation to interrupt another.

1.7. Unresolved Issues

We did not obtain solutions to all the Ironman requirements in the two-month period allotted to this design. In this section we sketch the way we would address the unresolved issues.

1.7.1. Machine-Dependent Code

Machine-dependent code presents two issues: definition of operations and definition of data. Tartan will permit separately-defined machine-dependent routines to be incorporated in the same way as other separate definitions. This is consistent with the Steelman requirement. We have not yet addressed the problem of machine-dependent declarations (data layout).

1.7.2. Simulation

We believe Tartan supports a programmed solution to the simulation requirement. For example, the facilities of Simula 60 can be provided for Tartan programs:

- Tartan activations can serve the same function as Simula activities.
- A coroutine call discipline may be programmed using the routines Activate and Suspend.
- A scheduler that manages simulated time can be programmed, again using operations on activations.

---

[1] The activation record itself may be allocated in the heap; it does not become eligible for garbage collection until all references have been broken. Thus no actname can become an uncontrolled pointer.

1.7.3. Definition of Integers

In the reference manual we chose fixed as a primitive and defined Int as a special case by choosing attributes appropriately. We believe it is possible to treat int as primitive and define Fixed as nonprimitive by associating range/precision bookkeeping with the operations.

1.7.4. Low-Level Input and Output

We included file as a primitive data type but did not specify its properties. Given the ability to write machine-dependent code to access the devices and the ability to use processes to maintain state (and hence to avoid, for example, re-opening a file for each operation), we believe a wide variety of low-level I/O can be implemented effectively.

1.7.5. Higher-Level Synchronization

Numerous synchronization disciplines have been proposed or are in active use. None of them clearly dominates the others; none is appropriate in all cases. We have elected to provide a very primitive synchronization tool, a latch. Conceptually, a latch is a spinlock; failure to sieze such a lock does not necessarily release the processor. By choosing a primitive mechanism, we hope to avoid pre-empting the implementation of higher-level synchronization techniques. We believe alternative mechanisms can be implemented effectively in Tartan. Indeed, we believe that this is the correct approach.

2. Programming Examples

Several sample Tartan programs are presented here. Some show the use of various features of the language; others provide programmed (nonprimitive) solutions to certain Ironman requirements.

2.1. Simple Static Data Type

A circular buffer is implemented in a vector. The definition is generic in the type of the elements; the length of the buffer is an attribute of the type. This implementation keeps a pointer to the current head of the buffer (Head) and a pointer to the element one past the current end of the buffer (Tail). All arithmetic on these pointers is done modulo the size of the buffer.

```
generic module CircularBuffers[T:type];
    begin
    exports CircBuf[T],                          ! type, attribute Size
        Clear, Append, Remove, Full, Empty,      ! routines
        BufOvfl;                                 ! exception

    type CircBuf[T](Size:Int) = record(Bf:array(0..Size-1) of T, Head,Tail: Int];

    exception BufOvfl;

    proc Clear(result C: CircBuf[T]); begin C.Head:=0; C.Tail:=0 end;

    proc Append(var C: CircBuf[T], const Val:T);
        begin
        if Full(C) then signal BufOvfl;
        C.Bf(C.Tail) := Val;
        C.Tail := mod(C.Tail+1, C.Size);
        end;

    proc Remove(var C:CircBuf[T], result Val: T);
        begin
        assert ¬ Empty(C);
        Val := C.Bf(C.Head);
        C.Head := mod(C.Head+1, C.Size);
        end;

    func Full(C:CircBuf[T])F:boolean; begin F := (C.Head = mod(C.Tail+1, C.Size)) end;

    func Empty(C:CircBuf[T])E:boolean; begin E := (C.Head = C.Tail) end;

    end    ! module CircularBuffers
```

2.2. Simple Dynamic Data Type

We define a list-processing module. Each list cell contains a value of a specific type; the definition of the module is generic in this type.

```
generic module ListDef[T:type];
    begin
    exports List[T], Data, Next,          ! type and field names
        Clear, Insert, Delete, Last;      ! routines

    type List[T] = dynamic record [Data:T, Next:List[T]];

    proc Clear(result L:List[T]); begin L := nil end;

    proc Insert(var Elt:List[T], Val:T);
        begin
        if Elt = nil
            then Elt := List[T]'(Val,nil)
            else Elt.Next := List[T]'(Val,Elt.Next)
        end;

    proc Delete(var Elt:List[T]); begin assert Elt ≠ nil; Elt := Elt.Next end;

    func Last(L:List[T])p:List[T];
        begin
        p := L;
        if p ≠ nil then while p.Next ≠ nil do p := p.Next od fi;
        end;

    end    ! module ListDef
```

2.3. Selecting Representations

Although Tartan treats types with different representations as different types, it is possible to use the variant and case facilities to define generic types that provide simiilar types with different representations. The representation is fixed during translation, when the generic definition is instantiated.

This example defines two alternative representations of queues. It has two generic parameters. The first is the type of the elements being queued, and it is used as in the previous examples. The second is a manifest constant, which is used to select which representation of queues is to be used. Since the variant is fixed during translation, there should be no loss of execution efficiency.

The two representations of queues are defined in terms of the circular buffers of section 2.1 and the lists of section 2.2.

```
generic module QueueDef[T:type, F:enum[Fix,Flex]];
    begin
    exports Queue[T],                          ! type, attribute Size
       Clear, Enq, Deq, Empty, Full,           ! routines
       QOvfl;                                  ! exception

    module Lst is ListDef[T];
    module CBf is CircularBuffers[T];

    type Queue[T](Size:Int) =
       variant manifest Fx: enum[Fix,Flex] := F
          [ on Fix -> CircBuf[T](Size) on Flex -> List[T] ]

    exception QOvfl;                    ! can only be raised on Queue[Fix]

    proc Clear(result Q: Queue[T]);
       begin
       case F on Fix -> Clear(Q(Fix)) on Flex -> Clear(Q(Flex)) esac
       end;

    proc Enq(var Q: Queue[T], const Val: T);
       begin
       case F
          on Fix  -> Append(Q(Fix), Val) { on BufOvfl -> signal QOvfl }
          on Flex -> Insert(Last(Q(Flex)), Val)
          esac
       end;

    proc Deq(var Q: Queue[T], result Val: T);
       begin
       case F
          on Fix  -> Remove(Q(Fix), Val)
          on Flex -> begin Val := Q(Flex).Data; Delete(Q(Flex)) end
          esac
       end;

    func Empty(Q:Queue[T])E:boolean;
       begin
       case F
          on Fix  -> E := Empty(Q(Fix))
          on Flex -> E := (Q(Flex) = nil)
          esac;

    func Full(Q:Queue[T])E:boolean;
       begin
       case F on Fix -> E := Full(Q(Fix).FixRep) on Flex -> E := false esac
       end;

    end    ! module QueueDef
```

2.4. Safe Data

Tartan does not provide indivisible operators for fetching and storing values. If parallel processes are operating, the programmer needs to take precautions to ensure the indivisibility of these operations. This program illustrates a solution that will work well with types for which fetching and storing the whole value makes sense.

```
begin

module Complex is assumed(ComplexLib);  ! Complex exports type Comp

generic module SafeData(T:type);
    begin
    exports Safe[T], Get, Put;          ! type name, fetch and store routines

    type Safe[T] = record (Lk:latch, Data:T);

    func Get(var S:Safe[T])R:T; begin Lock(S.Lk); R := S.Data; Unlock(S.Lk) end;

    proc Put(var S:Safe[T], var R:T); begin Lock(S.Lk); S.Data := R; Unlock(S.Lk) end;

    end;   ! module SafeData

module SafeComplex is SafeData(Comp);

var x,y,z: Safe[Comp];
Put(x, Comp'(1.,0.));
Put(y, Comp'(0.,1.));
Put(z, Get(x)+Get(y));

end;
```

Function Get takes a Safe[T] (here, a Safe[Comp]) as a var parameter. Since the Lk field is not exported from module SafeData, Get may use the procedures Lock and Unlock on that latch in order to protect the fetch.

Procedure Put specifies var parameters in both positions. Even though it does not alter R, a const specification would cause a copy.

The generic SafeData module is instantiated specifically for numbers of type Comp (the type exported by module Complex).

In the main program, the Comp constructor is used twice to generate values to store in the variables. The newly-constructed values in the calls on Put are accessible only in this program, so the constructor itself does not need to be indivisible. In the third assignment (call on Put), the addition is the addition for type Comp exported by module Complex.

3. Optional Additions to the Language

In the course of the Tartan design, we encountered a number of features that seemed attractive but could not be admitted because they violated either the Ironman requirement itself or the rule of minimality that we adopted for the design experiment. We list some of these here, indicating what they might add to the language and what they might cost.

Abbreviations for compound names. The import rule as stated can lead to the need for a substantial amount of qualification because all exported names, especially of types and routines, are potentially available pervasively. A renaming facility would reduce the need for explicit qualification. The renaming facility might involve renaming on import, or it might be a general with-clause. It would add convenience and probably improve the readability of the language. However, it would introduce a new construct in the language and introduce a new way to create aliases.

Less-than-global storage pools. As the language is defined, all dynamically allocated variables share the same heap. It would be possible to add the ability to declare a local sub-heap (zone) on the stack and allocate designated dynamic variables from it instead. There might be several zones active at once, with certain groups of variables sharing different ones. Alternatively, zones might be associated with blocks and all dynamic types defined in a block would share storage from a common zone. The cost is an additional mechanism and more complex scope rules. The benefit would be more control over dynamic variables and possibly more efficient storage recovery.

Resumable and parameterized exceptions An interrupt-style exception that has the semantics of a procedure call (resuming where it was raised) would be a useful thing to add. It would provide better control over many exception situations. Almost all the necessary mechanism must already be there to deal with the Notify command (i.e., the Terminate exception). In addition, the ability to pass parameters would be helpful, although it would complicate the syntax.

Richer control constructs. A loop exit and explicit function return could reduce the number of gotos and awkward conditional statements in programs. A richer collection of loop structures (downward counting, repeat with exitif, and so on) would add convenience. However, each such construct adds to the size of the language.

Assertions in declarations. As presently formulated, assertions are statements. It could be useful to permit them in declarations in order to check values of attributes and to guard initialization expressions. It would, however, require additional complexity in the syntax.

User-definable assignment. As noted in section 1.5.7, a default definition of assignment cannot anticipate all reasonable type definitions and all situations in which assignment makes sense. Only the programmer has the knowledge to do so. Tartan already permits infix operators to be overloaded for new types; there would be little additional cost for allowing ":=" to be overloaded as well.

References

[1] Department of Defense Requirements for High Order Computer Programming Languages, Revised "Ironman", July 1977. Appeared in SIGPlan Notices, 12, 12, December 1977 (pp. 39-54)

[2] Mary Shaw, Paul Hilfinger, Wm. A. Wulf, "TARTAN Language Design for the Ironman Requirement: Reference Manual", Carnegie-Mellon University Technical Report, June 1978.

ANHANG 8

6 JUNE 1978

SET OF SAMPLE PROBLEMS FOR PHASE II OF THE DESIGN CONTRACTS OF THE DOD HOL COMMONALITY EFFORT.

INTRODUCTION :

THIS SET OF SAMPLE PROBLEMS HAS BEEN SELECTED FROM A LARGER SET OF PROPOSALS MAINLY ON THE BASIS OF THE FOLLOWING CONSIDERATIONS:

- THE RESULTING PROGRAMS SHOULD BE LARGE ENOUGH TO ALLOW TO JUDGE THE 'APPEARANCE' AND THE 'READABILITY' OF PROGRAMS
- THEY SHOULD ALSO BE OF SUFFICIENT COMPLEXITY TO TEST INTERACTIONS BETWEEN LANGUAGE FEATURES
- AND, LAST BUT NOT LEAST, THEY SHOULD HAVE SOME RELATIONS TO ACTUAL APPLICATIONS.

AS TO THE AREAS TO BE INVESTIGATED, THE MAIN EMPHASIS WAS LAID UPON NOVEL LANGUAGE FEATURES, LIKE E.G. PARALLELISM, EXCEPTION HANDLING, AND NON-STANDARD I/O.

LIST OF SAMPLE PROBLEMS :

1 POLLED ASYNCHRONOUS INTERRUPT
2 PRIORITY INTERRUPT SYSTEM
3 A SMALL FILE HANDLING PACKAGE
4 DYNAMIC PICTURES
5 A DATABASE PROTECTION MODULE
6 A PROCESS CONTROL EXAMPLE
7 ADAPTIVE ROUTING ALGORITHM FOR A NODE WITHIN A DATA SWITCHING NETWORK
8 GENERAL PURPOSE REALTIME SCHEDULER
9 DISTRIBUTED PARALLEL OUTPUT
10 UNPACKING AND CONVERSION OF DATA

STRUCTURING OF EXAMPLES:

THE DESCRIPTION OF AN EXAMPLE CONTAINS :

1 A STATEMENT ON THE PURPOSE OF THE EXAMPLE
2 A DESCRIPTION OF THE PROBLEM TO BE SOLVED
3 ASSUMPTIONS ABOUT THE UNDERLYING CONFIGURATION
4 SOME GUIDELINES FOR THE SOLUTION

EX 1

POLLED ASYNCHRONOUS INTERRUPT

PURPOSE:

AN EXERCISE TO PROGRAM A DEVICE AND INTERRUPT HANDLER RELYING PRIMARILY UPON POLLING TECHNIQUES.

PROBLEM:

1 A CHANNEL HANDLER WILL EXPECT INPUT BY THE FUNCTION PROCEDURE CALL

'read(DEVICE-NUMBER)'

AND RETURN A CHARACTER FROM THAT DEVICES' INPUT-STREAM.

2 THEN SHOULD BE A MINIMUM DELAY FROM THE TIME A CHARACTER IS INTRODUCED INTO THE CIRCULAR BUFFER AND THE TIME IT MAY BE ACCESSIBLE BY A 'read'. (THE INPUT WILL BE DISPLAYED ON THE APPROPRIATE CRT BY THE readING PROCESS. APPARENT SIMULTANEITY OF HITTING THE KEY AND APPEARANCE ON THE CRT IS DESIRED, I.E. THE SYSTEM SHOULD BE REASONABLY EFFICIENT AND THUS PROVIDE GOOD RESPONSE-TIME.)
3 NO INPUT SHALL BE LOST.

ASSUMPTIONS

1 A 16-BT, BYTE ADRESSABLE MACHINE

2 AT LEAST 10 ASYNCHRONOUS INPUT DEVICES (KEYBOARDS) SHARING I/O CHANNEL0.

3 A HARD-WIRED CIRCULAR BUFFER OF 128 BYTES LOCATED AT BYTE-LOCATION 500(8). TWO POINTERS ARE PROVIDED IN CONJUNCTION WITH THE CIRCULAR BUFFER:
headpointer - A POINTER TO THE MOST RECENT INPUT
tailpointer - A POINTER TO THE TAIL OF THE CIRCULAR INPUT QUEUE

4 THE I/O CHANNEL WILL INITIALIZE BOTH THE HEAD- AND THE TAIL-POINTER TO THE SAME LOCATION WHEN THE SYSTEM IS RESET.

5 A DIFFERENCE IN THE CONTENTS OF THE HEAD- AND THE TAIL-POINTER INDICATES THAT INPUT HAS OCCURRED. MAINTENANCE OF THE HEAD-POINTER IS THE PROVINCE OF THE I/O CHANNEL. MAINTENANCE OF THE TAIL-POINTER IS THE PROVINCE OF THE CHANNEL HANDLER.

6 NO INTERRUPT SHALL OCCUR WHEN INPUT IS CLEARED EXCEPT AS NOTED IN 7 BELOW. THE HEAD-POINTER IS INCREMENTED AND THE INPUT STORED IN TWO BYTES SPECIFIED BY THE ADDRESS CONTAINED IN THE HEAD-POINTER.

7 AN INTERRUPT WILL OCCUR WHEN THE HEAD POINTER IS POINTING TO THE INPUT-ENTRY JUST BELOW THE ENTRY INDICATED BY THE TAIL POINTER TO INDICATE THAT PROCESSING MUST OCCUR TO PREVENT LOSS OF INPUT.

8 THE INTERRUPT LOCATION FOR CHANNEL 0 IS 440(8) AND IS TWO BYTES IN LENGTH TO SPECIFY THE LOCATION OF THE INTERRUPT HANDLING ROUTINE.

9 AN INTERRUPT CAUSES AN IMPLICIT call OF THE SPECIFIED ROUTINE. WHEN PROCESSING OF THE INTERRUPT HAS BEEN COMPLETED,A return WILL CAUSE THE INTERRUPTED PROCESS TO RESUME.

10 TO SIMPLIFY MATTERS,ASSUME
1) THE CONTEXT OF THE INTERRUPTED PROCESS IS AUTOMATICALLY SAVED AND RESTORED,THAT
2) NO PRIORITY INTERRUPT LEVELS NEED BE CONSIDERED;AND
3) NO CLEARING OF THE INTERRUPT IS REQUIRED.

3.5(REMARK)
EACH INPUT CONSISTS OF TWO BYTES:
BYTE 0 CONTAINS THE ascii CHARACTER
BYTE 1 CONTAINS THE DEVICE IDENTIFIER,0-9 TO IDENTIFY THE SENDING KEYBOARD

GUIDELINES

IT SHOULD BE TRIED TO FORMULATE THE PROGRAM AS HARDWARE-INDEPENDENT AS POSSIBLE AND CLEARLY SEPARATE THE INTERFACE TO THE HARDWARE-DEPENDENT INFORMATION.

EX 2

PRIORITY INTERRUPT SYSTEM

PURPOSE :

AN EXERCISE TO PROGRAM AN INTERRUPT KERNEL SUPPORTING FOUR LEVELS OF PRIORITY

PROBLEM :

AN INTERRUPT HANDLING MECHANISM SHALL BE DESCRIBED WITH THE FOLLOWING FUNCTIONAL CAPABILITIES:

1 HIGHER PRIORITY INTERRUPTS SHOULD BE ABLE TO PREEMPT LOWER PRIORITY INTERRUPT PROCESSES.

2 AS MUCH PROCESSING AS POSSIBLE SHOULD BE DONE WITH HIGHER PRIORITY INTERRUPS ENABLED. (REMARK: IN GENERAL, INTERRUPTS SHOULD ONLY BE DISABLED FOR THE SHORTEST POSSIBLE TIME)

3 A PROPER MECHANISM FOR THE RESUMPTION OF PROCESSING OF PREEMPTED LOWER LEVEL INTERRUPT(HANDLER)S MUST BE PROVIDED.

4 TO SIMPLIFY MATTERS THE BODY OF EACH INTERRUPT HANDLER MAY BE SIMULATED E.G. BY A COUNT OF THE INTERRUPTS FOR THAT PRIORITY LEVEL.

ASSUMPTIONS :

1 THERE ARE FOUR INTERRUPT PRIORITY LEVELS: 0,1,2,3.
THE LOWER THE NUMBER, THE HIGHER THE PRIORITY.

2 THERE IS AN INTERRUPT VECTOR LOCATED AT 20(8) WITH 4 BYTES FOR EACH PRIORITY LEVEL:
20(8):PRIORITY 0,24(8):P1,30(8):P2,34(8):P3

THESE LOCATIONS SPECIFY THE ADDRESS OF THE INTERRUPT HANDLER FOR THE CORRESPONDING PRIORITY LEVEL.

3 THE INTERRUPT ROUTINE IS INVOKED BY AN IMPLICIT CALL WHEN THE INTERRUPT OCCURS.
AT COMPLETION OF THE HANDLER'S PROCESSING, A return IS TO BE PERFORMED.

4 TO SIMPLIFY MATTERS, ASSUME THAT THE INTERRUPTED PROCESSES' CONTEXT IS AUTOMATICALLY SAVED AND RESTORED UPON call AND return .HOWEVER, THE INFORMATION CONCERNING THE ENABLEMENT AND DISABLEMENT OF INTERRUPTS IS NOT PART OF THE CONTEXT.

5 INTERRUPTS ARE ENABLED AND DISABLED WITH A 'SET INTERRUPT INSTRUCTION':

sin <OPERAND>.

THE INTERRUPTS TO BE ENABLED/DISABLED ARE SPECIFIED BY BITS 0-3 IN THE WORD ADDRESSED BY THE OPERAND. THE BIT FIELDS ARE:
BIT 0(LSB) : PRIORITY 0, BIT 1 : PRIORITY 1 , ETC.

THE VALUES OF THESE FIELDS ARE:
0 : DISABLE 1 : ENABLE

IN ORDER TO DISABLE ALL INTERRUPTS, PERFORM AN INSTRUCTION
sin DISABLE ALL, WHERE THE CONTENTS OF DA=0

6 NO CLEARING OF THE INTERRUPTS IS REQUIRED.

GUIDELINES :

SAME AS FOR EX1. IT SHOULD ALSO BE EASY TO REPLACE THE BODIES OF THE INTERRUPT-HANDLERS. (E.G. AT RUNTIME, TO ALLOW FOR FLEXIBLE REACTIONS TO AN INTERRUPT, ACCORDING TO CIRCUMSTANCES)

EX 3

A SMALL FILE HANDLING PACKAGE

PURPOSE :

AN EXERCISE TO SHOW HOW HIGHER-LEVEL I/O FUNCTIONS CAN BE CONSTRUCTED AND USED.

PROBLEM :

PROGRAM A FILE SYSTEM ACCORDING TO THE FOLLOWING SPECIFICATIONS:

1 FILES ARE BUILT BY PRODUCERS WHO CAN PERFORM THE FOLLOWING OPERATIONS:

```
create    ( FILENAME,ESTIMATED-SIZE)
write     ( FILENAME,DATA-AREA.)
endwrite  ( FILENAME )
```

THE DATA,CONTAINED IN 'DATA-AREA' ARE WRITTEN ON THE FILE WITH 'FILENAME'. 'DATA-AREA' CAN BE ANYTHING FROM A SINGLE VARIABLE TO AN ARRAY OF STRUCTURES IN MEMORY.

FILES ARE SEQUENTIAL,SO EACH WRITE ADDS A RECORD TO THE END. endwrite SIGNALS COMPLETION OF WRITING.

2 FILES ARE READ BY ONE OR MORE CONSUMERS WHO USE THE FOLLOWING OPERATION:

```
read ( FILENAME,RECORD-NO.,DATA-AREA )
```

HERE,DATA ARE READ FROM A GIVEN RECORD FROM FILE 'FILENAME'.

3 ONCE ALL READING IS COMPLETE,THE FILE MAY BE DESTROYED BY CALLING:

```
destroy ( FILENAME )
```

EXCEPTIONS SHALL BE RAISED IN AT LEAST THE FOLLOWING CASES:

A) IF A PRODUCER WANTS TO CREATE A FILE WITH AN ALREADY EXISTING FILENAME

B) IF A USER WANTS TO WRITE ON A NONEXISTENT FILE

C) IF A CONSUMER WANTS TO READ FROM A NONEXISTENT FILE OR FROM AN EXISTING FILE WITH A NONEXISTENT RECORD NUMBER

D) IF A FILE SHALL BE DESTROYED WHILE IT IS STILL USED BY SOMEBODY ELSE.

ASSUMPTIONS :

ASSUME A DISK AS STORAGE MEDIUM.

GUIDELINES :

THE DESIGN SHOULD PREVENT DEADLOCK OF FILE STORAGE, ALLOW DISK OPERATIONS TO BE SCHEDULED ACCORDING TO ANY SCHEDULE (WHERE THE SCHEDULER GOES, SHOULD BE INDICATED), AND PREVENT USERS FROM ACCESSING ANYTHING BUT THE ABOVE FIVE OPERATIONS.

EX 4

DYNAMIC PICTURES.

PURPOSE :

AN EXERCISE TO SHOW HOW A GRAPHIC DISPLAY OF A DYNAMIC SITUATION CAN BE PROGRAMMED.

PROBLEM :

ON A DISPLAY SCREEN A RECTANGULAR PATTERN OF E.G.10 HORIZONTAL AND 10 VERTICAL LINES SHALL BE DRAWN. (ONE MIGHT ALSO IMAGINE THAT THE BACKGROUND IS A SIMPLIFIED MAP.)

WITHIN THIS GRID TWO MOVABLE OBJECTS SHALL BE SHOWN. THEY SHALL BE DISCRIMINATED EITHER BY COLOR OR BY SHAPE.
THE SPEED AND DIRECTION OF EACH OBJECT SHALL BE CONTROLLED BY AN INPUT-DEVICE,E.G.A JOYSTICK.
THERE SHALL BE A RESET-BUTTON,WHICH ALLOWS TO BRING THE OBJECTS INTO SOME PREDEFINED POSITION AND A START-BUTTON,WHICH CAUSES THEM TO MOVE.
IF THE OBJECTS COLLIDE,THEY SHALL START TO BLINK AND,AFTER SOME SECONDS, RETURN TO THEIR HOMING-POSITION.THIS SHALL BE EQUIVALENT TO A reset .

ASSUMPTIONS :

THE 'start' AND THE 'reset' BUTTON SHALL BE CONNECTED TO THE INTERRUPT-HANDLING MECHANISM OF THE UNDERLYING SYSTEM IN A WAY THAT DIFFERENT INTERRUPTS OCCUR WHEN DIFFERENT BUTTONS ARE PRESSED.

THE CONTROLLING INPUT DEVICES SHALL BE PURELY PASSIVE,I.E.THE POSITION OF THE STICK (left,right,forward,reverse) AND ITS DEVIATION FROM 'POSITION ZERO', CONTROLLING THE SPEED OF THE OBJECTS,HAVE TO BE READ IN EXPLICITELY BY THE PROGRAM. THE POSITION OF THE INPUT-DEVICE SHALL BE ACCESSIBLE TO THE PROGRAM VIA TWO 16-BIT REGISTERS (TWO BYTES),ONE FOR EACH COORDINATE. EACH BYTE SHALL CONTAIN A SIX-BIT INTEGER NUMBER (RIGHT ADJUSTED) WHICH REPRESENTS THE DEFLECTION IN THIS PARTICULAR DIRECTION IN THE MOMENT OF READ-IN.
THERE EXIST ALL KINDS OF 'REASONABLE COMBINATIONS' OF THESE VALUES, E.G. 15right-60forward ,56left-10reverse .THE CONSTRUCTION OF THE HARDWARE SHALL BE SUCH THAT 'UNREASONABLE COMBINATIONS'CANNOT OCCUR,LIKE E.G. 10left-20right.

GUIDELINES :

THE HARDWARE CHARACTERISTICS OF THE DISPLAY-DEVICE WERE MAINLY LEFT OUT TO PREVENT THE SOLUTIONS FROM BECOMING TOO LENGTHY.

THE ALGORITHMS SHALL BE INDEPENDENT OF THE ACTUAL CHARACTERISTICS OF THE DISPLAY DEVICE,E.G.IT SHALL NOT MATTER WHETHER THE DISPLAY DEVICE

HAS A VECTOR GENERATOR OR WHETHER IT IS JUST ABLE TO PLOT RANDOM POINTS,
WHETHER THE OBJECTS CAN BE CREATED BY A PATTERN GENERATOR,
OR WHETHER THEY HAVE TO BE PUT TOGETHER FROM POINTS AND/OR LINES.
THE NECESSARY HARDWARE DEPENDENCIES SHOULD NEVERTHELESS BE CLEARLY IDENTIFIED AND AS WELL LOCALIZED AS POSSIBLE.
THE PROGRAM SHALL BE WRITTEN AND STRUCTURED IN A WAY THAT IT WILL WORK WITH THE MOST PRIMITIVE DISPLAY-HARDWARE, E.G. A RANDOM-POINT DISPLAY, WHICH HAS A PRECISION OF 10 BITS FOR EACH COORDINATE, BUT THAT THE ROUTINES NECESSARY FOR SIMULATING MORE COMPLEX DISPLAY CAPABILITIES CAN BE EASILY REMOVED.

TO SIMPLIFY MATTERS, IT CAN BE ASSUMED THAT THE LOWEST LEVEL OF OUTPUT-ROUTINES NEED NOT BE INCLUDED IN THE EXAMPLE, I.E. AS FAR AS THE PROBLEM IS CONCERNED, THE OUTPUT SHALL BE REGARDED AS COMPLETED, AS SOON AS THE COORDINATES OF POINTS (LINES, OBJECTS, E.T.C.) HAVE BEEN DEPOSITED AS INTEGER NUMBERS IN THE APPROPRIATE BUFFERS.

IT IS LEFT TO THE DESIGNER HOW HE CHOSES TO IMPLEMENT THE GRAPHIC REPRESENTATION, E.G. BY FORMATTING PROCEDURES (SIMILAR TO CHARACTER FORMATS) OPERATING ON BUILT-IN DATA TYPES OR BY SPECIAL DATA STRUCTURES.
IT IS ALSO LEFT TO HIM HOW HE WANTS TO IMPLEMENT THE EMERGENCY REACTION, EG. BY A SOFTWARE-INTERRUPT OR BY EXCEPTIONS.

EX 5

A DATABASE PROTECTION MODULE

PURPOSE :

AN EXERCISE TO DEMONSTRATE HOW COMPLEX SYNCHRONIZATION MECHANISMS CAN BE CONSTRUCTED ON USER LEVEL.

PROBLEM :

A DBMS SHALL CONTAIN A MODULE WHICH CONTROLS ACCESS TO GIVEN DATA AREAS.

THE USER (OR A RUNNING PROCESS) SHALL BE ABLE TO INDICATE WHETHER HE REQUIRES EXCLUSIVE ACCESS TO A CERTAIN PART OF A DATA BASE ('DATA-SET') OR WHETHER HE IS WILLING TO SHARE THIS RESOURCE WITH OTHER USERS (E.G.FOR READING).

THE RESPECTIVE OPERATIONS SHALL LOOK LIKE THE FOLLOWING:

```
exclusive (DATA-SET-NAME,PREEMPTION-PARAMETER);

shared (DATA-SET-NAME,PREEMPTION-PARAMETER);
```

BY THE FOLLOWING OPERATION THE USER SHALL BE ABLE TO INDICATE THAT HE NO LONGER WANTS TO USE THE DATA-SET:

```
free (DATA-SET-NAME);
```

IT SHALL BE POSSIBLE TO SPECIFY,EITHER BY AN EXECUTABLE STATEMENT AT ANY TIME OR BY A KIND OF DECLARATION AT SCOPE ENTRY OR AT COMPILE-TIME:

A) WHETHER AN EXCLUSIVE RESERVATION HAS PRIORITY OVER A SHARED RESERVATION

B) HOW MANY USERS MAY SHARE A RESOURCE
(THIS NUMBER MAY E.G.BE LIMITED BY THE LENGTH OF SOME WAITING QUEUES)

C) WHICH USERS MAY EXECUTE WHICH KIND OF ACCESS

D) WHETHER PREEMPTION IS POSSIBLE AND,IF NOT,WHETHER
AN EXCEPTION SHALL BE RAISED IN CASE OF AN ATTEMPT TO USE THE PREEMPTION PARAMETER.

E) WHETHER DIFFERENT USERS HAVE DIFFERENT PRIORITIES,AND,IF SO,WHICH ONES

F) WHETHER THE DEMANDING PROCESS SHALL JUST WAIT FOR THE AVAILABILITY OF THE DESIRED RESOURCE OR WHETHER IN THIS CASE AN EXCEPTION SHALL BE RAISED TO ALLOW FOR EVASIVE ACTION.

NOTE THAT 'USER' MAY IN THIS EXAMPLE ALSO ALWAYS MEAN :'RUNNING PROCESS'.

THE MODULE SHALL BE CODED IN THE COMPLETE FORM IT WOULD REQUIRE TO PUT IT INTO A LIBRARY.

PROPER PROCEDURES FOR CLEANUPS SHALL BE PROVIDED IN CASE OF PREEMPTION.

ASSUMPTIONS :

NO SPECIFIC ASSUMPTIONS AS FAR AS THE HARDWARE IS CONCERNED.

GUIDELINES :

IT IS THE IMPLEMENTOR'S OPTION WETHER HE PREFERS TO PROVIDE ONE VERY GENERAL MODULE WITH ALL THESE CAPABILITIES OR WETHER HE WANTS TO USE GENERIC FACILITIES TO CREATE MODULES WITH A PROPER SUBSET OF THE FUNCTIONALITIES DEPENDENT OF THE ACTUAL REQUIREMENTS AT THE POINT OF INSTANTIATION.

EX 6

A PROCESS CONTROL EXAMPLE

PURPOSE :

AN EXERCISE TO TEST INTERACTIONS BETWEEN PARALLEL PROCESSING AND EXCEPTION HANDLING.

PROBLEM :

ASSUME FOUR PROCESSES:

process a WHICH READS IN DATA FROM THE ENVIRONMENT AND STORES THEM IN A BUFFER AREA

process b WHICH PROCESSES THE DATA IT FINDS IN THE BUFFER AREA ACCORDING TO SOME ALGORITHM AND STORES THEM IN A 'RESULT AREA'.

process c WHICH PRODUCES OUTPUT AS A CONSEQUENCE OF THESE DATA (EITHER IN HUMAN-ORIENTED FORM OR AS CONTROL-OUTPUT FOR THE PROCESS TO BE CONTROLLED)

process d MONITORS AND CONTROLS THESE THREE (AND POSSIBLY OTHER) PROCESSES AND INTERACTS WITH THE OPERATOR VIA A KEYBOARD CONSOLE.

IT SHALL BE FURTHER ASSUMED THAT process a AND process b INTERACT IN THE FOLLOWING SPECIFIC WAY:

THE BUFFER IS ORGANIZED AS A 'DOUBLE-BUFFER',I.E.,AFTER ONE OF ITS TWO AREAS HAS BEEN FILLED BY process a,process b IS NOTIFIED AND STARTS TO READ OUT OF THE BUFFER. process a CONTINUES BY DEPOSITING DATA IN THE SECOND BUFFER AREA. IF THIS IS FULL,process a TRIES TO DEPOSIT DATA IN THE FIRST AREA AGAIN. process b,IN TURN,NOTIFIES process a AFTER HAVING READ ONE DATA AREA.

IT IS ILLEGAL TO READ A BUFFER AREA WHICH HAS NOT PREVIOUSLY BEEN FILLED AND TO WRITE INTO A BUFFER AREA WHICH HAS NOT BEEN COMPLETELY READ (EXCEPT IN THE INITIALIZATION PHASE).

THE PROGRAM SHALL BE STRUCTURED IN A WAY THAT IT IS POSSIBLE TO REPLACE process a BY APPROPRIATE HARDWARE WITHOUT HAVING TO CHANGE THE PROGRAM PARTS FOR PROCESSES b,c, AND d .

IT SHALL ALSO BE POSSIBLE TO TERMINATE process a AND b AT ANY TIME WITHOUT LOSING DATA,I.E.BEFORE TERMINATION A CLEANUP OPERATION SHALL BE INVOKED WHICH CAUSES PROCESSING OF ANY REMAINING DATA IN EITHER OF THE TWO BUFFER AREAS.

ASSUMPTIONS :

NO PARTICULAR ASSUMPTIONS AS FAR AS HARDWARE IS CONCERNED.

THE BUFFERS AND THE 'RESULT AREA' CAN BE ORGANIZED AS ARRAYS.

GUIDELINES :

TO SIMPLIFY MATTERS, IT CAN BE ASSUMED THAT ACTUAL INPUT-OUTPUT , I.E. THE COMMUNICATION WITH THE HARDWARE, AS WELL AS THE PROCESSING OF THE DATA IN process b IS DONE BY GIVEN LIBRARY ROUTINES.
THE ALGORITHM IN process d MAY ALSO BE DESCRIBED IN A HIGHLY SUMMARIZED FORM, BECAUSE THIS IS NOT WHAT THE EXAMPLE IS TO TEST.

EX 7

ADAPTIVE ROUTING ALGORITHM FOR A NODE WITHIN A DATA SWITCHING NETWORK

PURPOSE :

TEST FOR LANGUAGE SUITABILITY FOR MULTICOMPUTER AND COMMUNICATIONS APPLICATIONS.

PROBLEM :

Develop the program for a multiprocessor within one node of a data switching network to maintain the tables of
1) distances,
2) minimum delay time, and
3) routing for the following adaptive routing algorithm:

Each node in a network maintains a table of distances and a table of minimum delay times between itself and all other nodes. The distance metric is the minimum number of hops required to reach each other node. Both tables are maintained through updates in the form of table exchanges which occur only between neighbor nodes (nodes of distance, one). Each node maintains a routing table which directs routing through that neighbor node which achieves the minimum delay time.

In parallel with, and at the same periodic rate as this computing process, separate computing processes at each node are computing the minimum delay times to neighbors; and reading into computer memory the updated distance table of each neighbor, and the updated minimum delay time table of each neighbor. Initially each node knows only the distance to each neighbor, which is one, and the minimum delay time to each neighbor. Other distances and minimum delay times are initially considered infinite. Each node iteratively builds up its own distance and minimum delay time tables from the distance and minimum delay time tables exchanged with its neighbors, and updates tables containing such information about itself. Other computing processes transmit this information between such neighbors. Hence, the routing table at each node is established and periodically updated adaptively from the minimum delay times.

When a link is broken or established, a separate computing process at each of the two former or new neighbors, corrects the distance and minimum delay time tables.

The reason a distance table must be mined is that if the network is disconnected the algorithm causes the distance between disconnected nodes to increase without limit. Thus whenever the

distance between two nodes becomes greater than the number of nodes in the network, this distance and minimum delay time is considered infinite, and the node is considered unreachable.

In the example program, consider that the number of nodes in the network, the neighbors of the programmed node, and the periodic update interval are constants known at compile time.

ASSUMPTIONS :

NONE AS FAR AS THE HARDWARE IS CONCERNED.

GUIDELINES :

THE ACTUAL INTERCHANGE BETWEEN THE NODES CAN BE ASSUMED TO BE PERFORMED BY GIVEN LIBRARY ROUTINES

EX 8

GENERAL PURPOSE REAL-TIME SCHEDULER

PURPOSE :

AN EXERCISE TO TEST THE POSSIBILITIES FOR RELATING COMPUTATIONAL PROCESSES TO REAL TIME.

PROBLEM :

A LIBRARY MODULE SHALL BE WRITTEN WHICH ALLOWS TO SCHEDULE COMPUTATIONAL PROCESSES IN ACTUAL REAL TIME. THE NUMBER OF THESE PROCESSES SHALL BE VARYING, DETERMINABLE AT LINK-TIME.
THE SCHEDULER SHALL RECEIVE THE 'TICKS' OF THE REAL-TIME CLOCK OF THE SYSTEM (E.G. BY REACTING TO THE RESPECTIVE INTERRUPT) AND TRANSFORM THEM INTO ACTUAL REAL TIME, E.G. BY APPLYING THE PROPER COMPILE-TIME CONSTANTS.

TO SIMPLIFY MATTERS, THE TIME SPAN WHICH CAN BE HANDLED BY THE SCHEDULER, MAY BE RESTRICTED TO 24 HOURS, I.E. ALL TIMES WILL BE COMPUTED MODULO 24 HOURS.

THIS 'REAL TIME' SHALL BE ACCESSIBLE TO THE PROGRAM BY THE COMMAND

```
time (OPERAND)
```

WHICH SHALL DEPOSIT THE TIME (AT THE POINT IN TIME THE OPERATION IS EXECUTED) IN THE LOCATION INDICATED BY 'operand' AS AN ascii CHARACTER STRING WITH THE FOLLOWING CONVENTIONS:

FIRST TWO CHARACTERS:	HOURS
SECOND TWO CHARS :	MINUTES
THIRD TWO CHARACTERS:	SECONDS

BUT THE MAIN PURPOSE OF THE SCHEDULER SHALL BE THE INITIATION OF THE EXECUTION OF COMPUTATIONAL PROCESSES ACCORDING TO PREDEFINED CONDITIONS IN REAL TIME. THIS SHALL BE POSSIBLE EITHER ONCE OR REPEATEDLY.

PROCESSES SHALL BE CONNECTED TO THE SCHEDULER BY OPERATIONS OF THE FORM:

```
execute PROCESSNAME,TIME
execute TIME /*MEANING THE PROCESS WHICH PERFORMS THIS OPERATION*/
execute PROCESSNAME,START-TIME,REPETITION-INTERVAL
```

INTENTIONALLY NO EXACT REPRESENTATION FOR THESE OPERATIONS IS GIVEN IN THE EXAMPLE (ESPECIALLY IT SHALL NOT BE IMPLIED THAT THEY ARE PROCEDURE CALLS). THE REPRESENTATION SHALL BE PROPOSED BY THE LANGUAGE DESIGNER IN ORDER TO :
1) FIT INTO THE TEXT OF A USER PROGRAM AS SIMPLY AND NATURALLY AS POSSIBLE AND
2) BE EFFICIENTLY IMPLEMENTABLE IN THE LANGUAGE PROPOSED.

IF TWO PROCESSES ARE DUE FOR EXECUTION AT THE SAME POINT IN TIME, THEY SHALL BE ACTIVATED IN PRIORITY ORDER.

NOTE, THAT IN ORDER TO ACHIEVE THIS, A LIBRARY ROUTINE MAY HAVE TO BE USED, WHICH SORTS THE CONTROL BLOCKS OF THE SCHEDULED PROCESSES ACCORDING TO THEIR PRIORITY. BECAUSE SUCH A SORTING ROUTINE IS OF GENERAL INTEREST, IT SHOULD ALSO BE USEABLE FOR OTHER DATA-TYPES. IT SHOULD BE DEMONSTRATED, HOW THE PARAMETER PASSING MECHANISM OF SUCH A ROUTINE IS FIT FOR THIS PURPOSE WITHOUT CAUSING TOO MUCH RUNTIME OVERHEAD.
FOR THE PURPOSE OF THE EXAMPLE THE SORTING ALGORITHM PROPER MAY BE SIMPLE AND INEFFICIENT, BECAUSE IT IS NOT RELEVANT FOR THE DEMONSTRATION.

IT MUST ALSO BE POSSIBLE TO DISCONNECT PROCESSES FROM THE SCHEDULER AT ANY POINT IN TIME, EITHER BY ACTION FROM THEMSELVES OR FROM OTHER PROCESSES.

ASSUMPTIONS :

ASSUME A SYSTEM CLOCK WHICH DELIVERS 'TICKS' OF A FREQUENCY WHICH IS SUFFICIENT TO DO THE NECESSARY COMPUTATIONS WITH THE NECESSARY PRECISION.
THE WAY, HOW PROCESSES CAN BE MADE KNOWN TO THE SCHEDULER, DEPENDS ON THE IMPLEMENTATION MODEL, WHICH UNDERLIES THE LANGUAGE PROPOSAL.

EX 9

DISTRIBUTED PARALLEL OUTPUT :
==========

PURPOSE :

AN EXERCISE TO DEMONSTRATE THE ABILITY OF PROCESSING PARALLEL EVENTS WHICH NEED NOT PROGRESS AT THE SAME RATE.

PROBLEM :

THIS PROGRAM HAS ENCOUNTERED A MULTIPLE ADDRESSEE MESSAGE TO BE OUTPUT OVER A NUMBER OF ASYNCHRONOUS LINKS.
EACH LINK IS CONTROLLED BY AN INDIVIDUAL PROCESS WHICH PERFORMS ALL LINK RELATED PROCESSING. EACH PROCESS CAN ACCEPT ONE PACKET OF THE MESSAGE AT A TIME AND WILL NOTIFY THE PROGRAM WHEN THE LAST PACKET FURNISHED TO IT HAS BEEN ACKNOWLEDGED BY THE DISTANT STATION.
WHEN ALL TRANSMISSIONS ARE COMPLETE, THE PROGRAM SHALL PURGE THE MESSAGE.

ASSUMPTIONS :
---

1 THE MESSAGE HAS FIVE ADDRESSEES, BUT THESE CAN BE DIFFERENT FOR EACH MESSAGE.

2 THE MESSAGE IS FIVE PACKETS LONG.

3 EACH PACKET IS 80 BYTES LONG.

4 THE BUFFERS CONTAINING THE MESSAGE ARE CONTIGUOUSLY LOCATED.

5 AT INITIALIZATION THE PROGRAM SHALL BE FURNISHED THE ADDRESS OF THE FIRST BUFFER, THE NUMBER OF BUFFERS, AND THE IDENTITY OF THE FIVE LINKS OVER WHICH THE MESSAGE IS TO BE SENT (EACH LINK IS CONTROLLED BY AN INDIVIDUAL PROCESS, NAMED L0..L9).
THE LINK IDENTIFICATION SHALL BE IN THE FORM (Ln,Ln,Ln...)WHERE N HAS LEGAL VALUES BETWEEN 0 AND 9.

6 AN 8 BIT MACHINE (ONE OF TODAY'S TYPICAL MICROPROCESSORS)

7 THE PROGRAM WILL BE CAPABLE OF PROCESSING UP TO TEN ADDRESSEES.

8 THERE IS NO QUEUING DELAY, I.E. THE LINK-PROCESSES ARE DEDICATED AND CAN REACT IMMEDIATELY.

remark : ONE CAN ASSUME THAT THE INDIVIDUAL LINK PROCESSES ARE RESIDENT IN DEDICATED MICROPROCESSORS AND THAT THE COORDINATION IS DONE IN ANOTHER PROCESSOR TO WHICH THEY ARE CONNECTED BY A BUS.

GUIDELINES :

NONE.