

inmos

TRANSPUTER DEVELOPMENT AND *iq* SYSTEMS DATABOOK

First Edition 1989

INMOS Databook Series

Transputer Databook

Military Micro-products Databook

Transputer Development and *iq* Systems Databook

Memory Databook

Graphics Databook


Digital Signal Processing Databook

Transputer Applications Notebook: Architecture and Software

Transputer Applications Notebook: Systems and Performance

Copyright ©INMOS Limited 1989

INMOS reserves the right to make changes in specifications at any time and without notice. The information furnished by INMOS in this publication is believed to be accurate; however, no responsibility is assumed for its use, nor for any infringement of patents or other rights of third parties resulting from its use. No licence is granted under any patents, trademarks or other rights of INMOS.

 **Inmos**, IMS and OCCAM are trademarks of INMOS Limited.

INMOS is a member of the SGS-THOMSON Microelectronics Group.

INMOS document number: 72 TRN 219 00

Contents overview

1	Systems products overview	1
	BOARDS	5
2	TRAnsputer Modules (TRAMs)	7
	Transputer plus memory:	
	IMS T222 16-bit transputer:	
	64 Kbytes, Size 1, B416	8
	IMS T414, T425 and T800 32-bit transputers:	
	32 Kbytes, Size 1, B401	14
	1 Mbyte, Size 1, B411	20
	2 Mbytes, Size 2, B404	26
	4 Mbytes, Size 4, B417	33
	8 Mbytes, Size 8, B405	40
	IMS T801 32-bit transputer:	
	160 Kbytes, Size 2, B410	47
	Application specific TRAMs:	
	Differential link buffer TRAM Size 2, B415	53
	Flash ROM TRAM, Size 2, B418	54
	Ethernet TRAM, Size 8, B407	56
	IEEE 488 GPIB TRAM, Size 4, B421	65
	SCSI TRAM, Size 2, B422	66
	Frame store TRAM, Size 8, B408	67
	Display TRAM, Size 8, B409	75
	Integrated graphics TRAM, Size 6, B419	84
	Vector processor TRAM, Size 4, B420	94
3	Standard interface boards	97
	IBM PC, 10 slots for TRAMs, B008	98
	VMEbus master, T800, 2Mbytes, 2 slots for TRAMs, B011	102
	VMEbus slave, 8 slots for TRAMs, B014	105
	VMEbus full master, slave, T801, 4Mbytes, B016	108
	NEC 9800 series, 5 slots for TRAMs, B015	110
	Double extended eurocard, 16 slots for TRAMs, B012	118
4	Evaluation boards	131
	Disk board, M212, 20 Mbyte hard disk, floppy, B005	132
	Signal processing board, 4x A100, IBM PC bus, B009	134

DEVELOPMENT SYSTEMS	139
5 Software development tools	141
Occam 2 toolset product overview	142
Parallel C compiler product overview	152
Parallel Fortran compiler product overview	158
Pascal compiler product overview	164
ADA compilers product overview	165
IMS D700 Transputer Development System	168
6 Board support software	175
IMS S708 and IMS S514 product overview	176
Ethernet support software product overview	179
7 Transputer development kits	183
Transputer introduction kit	184
Transputer performance evaluation kit	184
Custom development kits	184
IMS B211 INMOS transputer evaluation module (ITEM)	186
APPLICATIONS	189
8 Dual-in-line transputer modules (TRAMs)	191
9 Module motherboard architecture	209
10 Some issues in scientific language application porting and farming using transputers	233
11 Using the D705B occam toolset with non-occam applications	299
APPENDICES	
A Quality and Reliability	367
B Cables for Board Products	371

Contents

Preface		xviii
1	Systems products overview	1
1.1	Introduction	2
1.2	Innovation and Quality	2
1.3	TRAMS (TRAnsputer Modules)	3
1.3.1	Standard Interface	3
1.3.2	Upgradeability	3
1.3.3	Flexibility	3
1.3.4	Evaluation	4
1.4	Quality and Reliability	4
1	Boards	5
2	TRAnsputer Modules (TRAMs)	7
2.1	IMS B416 TRAM engineering data	8
2.1.1	Introduction	9
2.1.2	Pin descriptions	9
2.1.3	Standard TRAM signals	9
	notError (pin 11)	9
	LinkSpeedA and LinkSpeedB (pins 6 and 7)	9
	Link signals	10
2.1.4	Memory configuration	10
2.1.5	Mechanical details	11
2.1.6	Installation	12
2.1.7	Specification	13
2.1.8	Ordering Information	13
2.2	IMS B401 TRAM engineering data	14
2.2.1	Introduction	15
2.2.2	Pin descriptions	15
2.2.3	Standard TRAM signals	16
	notError (pin 11)	16
	LinkSpeedA and LinkSpeedB (pins 6 and 7)	16
	Link signals	16
2.2.4	Memory configuration	16
	Location of external memory	16
2.2.5	Mechanical details	17
2.2.6	Installation	18
2.2.7	Specification	19
2.2.8	Ordering Information	19
2.3	IMS B411 TRAM engineering data	20
2.3.1	Description	21
2.3.2	Pin descriptions	21
2.3.3	Standard TRAM signals	21
	notError (pin 11)	21
	LinkSpeedA and LinkSpeedB (pins 6 and 7)	22
	Link signals	22
2.3.4	Memory configuration	22
2.3.5	Mechanical details	23

	2.3.6	Installation	24
	2.3.7	Specification	25
	2.3.8	Ordering Information	25
2.4		IMS B404 TRAM engineering data	26
	2.4.1	Introduction	27
	2.4.2	Pin descriptions	27
	2.4.3	Standard TRAM signals	28
		notError (pin 11)	28
		LinkSpeedA and LinkSpeedB (pins 6 and 7)	28
		Link signals	28
	2.4.4	Subsystem signals	28
	2.4.5	Memory configuration	28
		Location of external memory	29
		Subsystem register locations	29
	2.4.6	Mechanical details	30
	2.4.7	Installation	31
	2.4.8	Specification	32
	2.4.9	Ordering Information	32
2.5		IMS B417 TRAM engineering data	33
	2.5.1	Introduction	34
	2.5.2	Pin descriptions	34
	2.5.3	Standard TRAM signals	35
		notError (pin 11)	35
		LinkSpeedA and LinkSpeedB (pins 6 and 7)	35
		Link signals	35
	2.5.4	Subsystem signals	35
	2.5.5	Memory configuration	36
		Location of external memory	36
		Subsystem register locations	36
	2.5.6	Mechanical details	37
	2.5.7	Installation	38
	2.5.8	Specification	39
	2.5.9	Ordering Information	39
2.6		IMS B405 TRAM engineering data	40
	2.6.1	Introduction	41
	2.6.2	Pin descriptions	41
	2.6.3	Standard TRAM signals	41
		notError (pin 11)	41
		LinkSpeedA and LinkSpeedB (pins 6 and 7)	42
		Link signals	42
	2.6.4	Subsystem signals	42
	2.6.5	Memory configuration	42
		Location of external memory	42
		Subsystem register locations	42
		Memory parity	43
	2.6.6	Installation	44
	2.6.7	Mechanical details	44
	2.6.8	Specification	46
	2.6.9	Ordering Information	46
2.7		IMS B410 TRAM engineering data	47
	2.7.1	Description	48
	2.7.2	Pin descriptions	48
	2.7.3	Standard TRAM signals	49

	notError (pin 11)	49
	LinkSpeedA and LinkSpeedB (pins 6 and 7)	49
	Link signals	49
	2.7.4 Memory configuration	49
	2.7.5 Mechanical details	50
	2.7.6 Installation	51
	2.7.7 Specification	52
	2.7.8 Ordering Information	52
2.8	IMS B415 TRAM product overview	53
2.9	IMS B418 flash-ROM TRAM product overview	54
	2.9.1 Specification	55
	2.9.2 Ordering Information	55
2.10	IMS B407 TRAM engineering data	56
	2.10.1 Transputer Modules (TRAMs)	57
	2.10.2 Pin descriptions	57
	2.10.3 Ethernet Capabilities	58
	Connecting to Ethernet (10BASE5)	58
	Connecting to Cheapernet (10BASE2)	60
	2.10.4 Memory Map	61
	2.10.5 Using the IMS B407	61
	2.10.6 Mechanical details	62
	2.10.7 Installation	62
	2.10.8 Specification	64
	2.10.9 Ordering Information	64
2.11	IMS B421 GPIB TRAM product overview	65
2.12	IMS B422 SCSI TRAM product overview	66
2.13	IMS B408 TRAM engineering data	67
	2.13.1 Introduction	68
	2.13.2 Pin descriptions	68
	2.13.3 Pixel Port signals	69
	Electrical Specification	71
	2.13.4 Memory Map	71
	2.13.5 Pixel Port control registers	71
	2.13.6 Mechanical details	72
	2.13.7 Installation	72
	2.13.8 Specification	74
	2.13.9 Ordering Information	74
2.14	IMS B409 TRAM engineering data	75
	2.14.1 Introduction	76
	2.14.2 Pin descriptions	76
	2.14.3 Pixel Bus connectors	77
	2.14.4 The Pixel channels	78
	8 bits/pixel mode	79
	18 bits/pixel mode	79
	The colour look-up tables	79
	Video Outputs	79
	2.14.5 Memory Map	79
	Pixel Channel Mode select	79
	The video timing generator	80
	The Colour look-up tables	80
	2.14.6 Mechanical details	81
	2.14.7 Installation	81
	2.14.8 Specification	83

	2.14.9	Ordering Information	83
2.15		IMS B419 TRAM engineering data	84
	2.15.1	Introduction	85
	2.15.2	Screen sizes	85
	2.15.3	Pin descriptions	86
	2.15.4	Memory Map	88
		SubSystem registers	88
	2.15.5	IMS G300 clock selection	89
	2.15.6	Jumper selection	90
	2.15.7	Video and sync outputs	90
	2.15.8	Mechanical details	91
	2.15.9	Installation	91
	2.15.10	Specification	93
	2.15.11	Ordering Information	93
2.16		IMS B420 VECTRAN product overview	94
	2.16.1	Specification	95
	2.16.2	Ordering Information	95
3		Standard interface boards	97
3.1		IMS B008 IBM PC Module Motherboard product overview	98
	3.1.1	Product Overview	99
	3.1.2	TRAM Slots	99
	3.1.3	System Services	99
	3.1.4	Link Configuration	99
	3.1.5	IBM PC Bus Interface	99
		Interrupts	100
		DMA	100
	3.1.6	Link Speeds	100
	3.1.7	Technical Summary	101
	3.1.8	Ordering Information	101
3.2		IMS B011 Tranputer VMEbus Master Card product overview	102
	3.2.1	Processor	103
	3.2.2	Booting	103
	3.2.3	Interrupts	103
	3.2.4	Memory	103
	3.2.5	VMEbus Interface	103
	3.2.6	RS232 ports	104
	3.2.7	TRAM slots	104
	3.2.8	Ordering Information	104
3.3		IMS B014 VMEbus Module Motherboard product overview	105
	3.3.1	VMEbus Interface	106
	3.3.2	Interrupts	106
	3.3.3	IMS C004 Control	106
	3.3.4	System Services Organisation	107
	3.3.5	Technical Summary	107
	3.3.6	Ordering Information	107
3.4		IMS B016 VMEbus master/slave Motherboard product overview	108
	3.4.1	General description	109
	3.4.2	Ordering Information	109
3.5		IMS B015 Module Motherboard product overview	110
	3.5.1	Link connections	111
	3.5.2	Link speed selection	111
	3.5.3	System Services	112

	3.5.4	Up, Down, and Subsystem	112
	3.5.5	PC interface	113
	3.5.6	IO Address	113
	3.5.7	Reset, Analyse and Error registers	113
	3.5.8	Interface link	113
		Interrupts	114
	3.5.9	External power supplies	115
	3.5.10	External Connections	116
	3.5.11	Specification	117
	3.5.12	Ordering Information	117
	3.6	IMS B012 Double Eurocard Motherboard engineering data	118
	3.6.1	Introduction	119
	3.6.2	Hardware Description	119
		Link Connections	119
		P1 Links	122
		Switch Configuration Transputer	124
		Reset, Analyse and Error	124
		Link Termination	126
		Error Lights	127
		User Power Connector	128
		Uncommitted Pins	128
	3.6.3	Ordering Information	129
4		Evaluation boards	131
	4.1	IMS B005 Double Extended Eurocard product overview	132
	4.1.1	Ordering Information	133
	4.2	IMS B009 DSP System Evaluation Board product overview	134
	4.2.1	The IMS B009 Evaluation Board	135
	4.2.2	Board Description	136
	4.2.3	Programming	137
	4.2.4	Product summary	137
	4.2.5	Technical summary	138
	4.2.6	Ordering details	138
2		Development systems	139
	5	Software development tools	141
	5.1	occam 2 toolset product overview	142
	5.1.1	Product overview	143
		occam 2 development system	143
		Support for mixed language developments	144
		System building and program consistency	144
		Source level debugging tools	144
		Support for teams of developers	145
	5.1.2	occam 2 toolset product description	145
		Documentation	145
		Software tools	146
		Software libraries	147
		Programming examples	148
	5.1.3	D700D transputer development system support	148
	5.1.4	occam 2 toolset product components summary	149
		Documentation	149
		Software tools	149

	Software libraries	149
5.1.5	D705 IBM PC version	150
	Operating requirements	150
	Distribution media	150
5.1.6	D605 VAX VMS version	150
	Operating requirements	150
	Distribution media	150
5.1.7	D505 SUN 3 version	150
	Operating requirements	150
	Distribution media	151
5.1.8	Associated products	151
5.1.9	Licencing information	151
5.1.10	Error reporting and field support	151
5.2	Parallel C compiler product overview	152
5.2.1	Product overview	153
	Support for parallelism	153
	Using C with the occam 2 toolset	153
5.2.2	3L Parallel C description	153
	Documentation	153
	Software tools	154
	Software libraries	155
5.2.3	3L C components summary	155
	Documentation	155
	Software tools	155
	Software libraries	155
5.2.4	D711 IBM PC version	156
	Operating requirements	156
	Distribution media	156
5.2.5	D611 VAX VMS version	156
	Operating requirements	156
	Distribution media	156
5.2.6	D511 SUN 3 version	156
	Operating requirements	156
	Distribution media	157
5.2.7	Associated products	157
5.2.8	Licencing information	157
5.2.9	Error reporting and field support	157
5.3	Parallel FORTRAN compiler product overview	158
5.3.1	Product overview	159
	Support for parallelism	159
	Using FORTRAN with the occam 2 toolset	159
5.3.2	3L Parallel FORTRAN description	159
	Documentation	159
	Software tools	160
	Software libraries	161
5.3.3	3L FORTRAN components summary	161
	Documentation	161
	Software tools	161
	Software libraries	162
5.3.4	D713 IBM PC version	162
	Operating requirements	162
	Distribution media	162
5.3.5	D613 VAX VMS version	162

	Operating requirements	162
	Distribution media	162
5.3.6	D513 SUN 3 version	163
	Operating requirements	163
	Distribution media	163
5.3.7	Associated products	163
5.3.8	Licencing information	163
5.3.9	Error reporting and field support	163
5.4	Pascal Compiler product overview	164
5.5	Ada Compilers product overview	165
	5.5.1 Ada Compilers for the Transputer	166
	5.5.2 Features	166
	5.5.3 Recommended Configuration	167
	Recommended Configuration for PC mothered compiler	167
	Recommended Configuration for VAX hosted compiler	167
5.6	IMS D700 Transputer Development System	168
	5.6.1 Product overview	169
	The user interface	169
	occam 2 compiler	169
	Loading programs into transputer networks	170
	Debugging	170
	5.6.2 Product description	170
	Documentation	170
	Software components	171
	5.6.3 Product components	172
	Documentation	172
	Integrated software components	172
	Software libraries	173
	5.6.4 D705B occam 2 toolset support	173
	5.6.5 Operating requirements	173
	5.6.6 Distribution media	173
	5.6.7 Licencing information	173
	5.6.8 Error reporting and field support	174
6	Board support software	175
	6.1 IMS S708 and IMS S514 product overview	176
	6.1.1 Product overview	177
	Support for other hosts	177
	6.1.2 Product components summary	177
	Documentation	177
	Software tools	177
	6.1.3 IMS S708	178
	Operating requirements	178
	Distribution media	178
	6.1.4 IMS S514	178
	Operating requirements	178
	Distribution media	178
	6.2 Ethernet Support Software product overview	179
	6.2.1 Product overview	180
	6.2.2 Product description	180
	6.2.3 Software components	180
	6.2.4 Hardware requirements	180
	6.2.5 Compatibility considerations	180

	6.2.6	Performance	180
	6.2.7	User Documentation	181
	6.2.8	Distribution media	181
	6.2.9	Related products	181
7		Transputer development kits	183
	7.1	Transputer Introduction Kit	184
	7.2	Transputer Performance Evaluation Kit	184
	7.3	Custom Development Kits	184
	7.4	IMS B211 INMOS Transputer Evaluation Module (ITEM)	186
	7.4.1	Introduction	187
	7.4.2	Applications	187
	7.4.3	Rear Connector Panel	187
	7.4.4	FCC Compliance	187
	7.4.5	Ordering Information	187
3		Applications	189
8		Dual inline transputer modules (TRAMs)	191
	8.1	Background	192
	8.2	Introduction	193
	8.3	Functional description	194
	8.3.1	Pinout of size1 module	194
	8.3.2	Pinout of larger sized modules	194
	8.3.3	TRAMs with more than one transputer	196
	8.3.4	Extra pins	196
	8.3.5	Subsystem signals driven from a TRAM	196
	8.3.6	Memory parity	198
	8.3.7	Memory map	198
	8.4	Electrical description	199
	8.4.1	Link outputs	199
	8.4.2	Link inputs	199
	8.4.3	notError output	199
	8.4.4	Reset and analyse inputs	199
	8.4.5	Clock input	200
	8.4.6	notError input to subsystem	200
	8.4.7	GND, VCC	200
	8.5	Mechanical description	200
	8.5.1	Width and length	200
	8.5.2	Vertical dimensions	201
	8.5.3	Direction of cooling	203
	8.6	TRAM pins and sockets	203
	8.6.1	Stackable socket pin	203
	8.6.2	Through-board sockets	203
	8.6.3	Subsystem pins and sockets	204
	8.6.4	Motherboard sockets	204
	8.7	Mechanical retention of TRAMs	204
	8.8	Profile drawings	205

9	Module motherboard architecture	209
9.1	Introduction	210
9.2	Module motherboard architecture	210
9.2.1	Design goals	210
9.2.2	Architecture	210
9.3	Link configuration	211
9.3.1	Pipeline	211
9.3.2	IMS C004 link configuration	212
9.3.3	T212 pipeline and C004 control	212
9.3.4	Software link configuration	212
9.4	System control	214
9.4.1	Reset, analyse and error	214
9.4.2	Up, down and subsystem	214
9.4.3	Source of control	216
9.4.4	Clock	221
9.5	Interface to a separate host	221
9.5.1	Link interface	221
9.5.2	System control interface	222
9.5.3	Interrupts	223
9.6	Mechanical considerations	223
9.6.1	Dimensions	223
	Width and length	223
	Vertical dimensions	224
9.6.2	Motherboard sockets	225
9.6.3	Mechanical retention of TRAMs	225
9.6.4	Module orientation	226
9.7	Edge connectors	226
10	Some issues in scientific language application porting and farming using transputers	233
10.1	Introduction	234
10.1.1	Background	234
10.1.2	Document notes	234
10.2	Preliminary information	234
10.2.1	Transputers	235
10.2.2	Processes	235
10.2.3	The transputer / host development relationship	236
10.2.4	Why port to a transputer?	236
10.2.5	Different categories of application porting	237
	Transputer software development tools	238
10.3	Altering the application as little as possible	238
10.3.1	The scenario	238
10.3.2	Suitable applications	240
	Requirements	240
	Good candidates	240
10.3.3	Identifying the best transputer for your application	240
10.3.4	Some potential porting difficulties	241
10.3.5	An implementation overview	241
10.3.6	Porting example : SPICE	242
	About SPICE	242
	Performance	242
10.3.7	Porting example : T _E X	243

	About T_EX	243
	Performance	243
	10.3.8 Further work	244
10.4	Parallelizing the application	244
	10.4.1 Types of parallelism	244
	10.4.2 Why parallelize ?	245
	10.4.3 Definitions	245
	10.4.4 The stages in modularizing	246
	10.4.5 Modules	246
	Module properties	247
	Modules provided by the INMOS tools	247
	Instancing modules	248
	Module structure	249
	Module communication requirements	249
	Module communication protocol	249
	10.4.6 Guidelines on dividing an application into modules	250
10.5	Implementing modules	252
	10.5.1 The technique	252
	Overview	252
	Benefits	253
	10.5.2 Example of module implementation	254
	10.5.3 Implementation notes	257
	10.5.4 Some coding examples	261
	10.5.5 Software methods of increasing performance	265
	Good ideas	265
	Bad ideas	267
	10.5.6 Further work	269
10.6	Using transputers with other processors	269
	10.6.1 Suitable applications	271
	10.6.2 Software support for mixed processor systems	272
	Accommodating architectural differences	272
	Using services provided by another processor	272
	10.6.3 Hardware support for mixed processor systems	273
	10.6.4 Communication mechanisms	274
	Communication by explicit polling	274
	Communication by explicit DMA	277
	Communication by device drivers	277
	Increasing data exchange bandwidth by software means	279
	10.6.5 Implementation strategy	280
	10.6.6 Testing strategy	281
	10.6.7 Further work	281
	10.6.8 Mixed processor example	282
10.7	Farming an application	283
	10.7.1 Suitable applications	285
	10.7.2 General farm discussion	285
	The software components	285
	The farm protocol	285
	10.7.3 Interfacing to the farm	286
	Interfacing to another transputer process	286
	Interfacing to a process on a non-transputer processor	286
	10.7.4 Performance issues	287
	Linearity	287
	Priority	287

	Protocol	287
	Overheads	288
	Buffering	288
	Load balancing	288
	General farming principles	288
10.7.5	Farming part of an application	289
	Scenario	289
	Implementation	290
10.7.6	Farming an entire application	290
	Scenario	290
	Implementation	290
	Alternative implementation	291
10.7.7	Farming a heterogeneous processor application	291
	Scenario	291
	Implementation	291
	Alternative implementation	292
10.7.8	Part port farm example : Second Sight	293
	About Second Sight	293
	Performance	293
10.7.9	Further work	293
	Flood-filling a transputer network	293
	Extraordinary use of transputer links	294
	Overcoming i/o bottlenecks	294
	Comparison between farms and application pipelining	295
	Farms of farms	295
	Dynamic link switching	295
10.8	Planning the structure of a new application	295
10.9	Summary and Conclusions	296
10.10	References	297
11	Using the D705B occam toolset with non-occam applications	299
11.1	Introduction	300
	11.1.1 Article notes	300
11.2	Background information	300
	11.2.1 Transputers	300
	11.2.2 The transputer / host development relationship	300
	11.2.3 Connecting transputers together	301
	11.2.4 The other occam toolsets	302
11.3	The INMOS scientific-language compilers	302
	11.3.1 The compilers	302
	Features	303
	11.3.2 Using the scientific-language compilers in the simplest case	303
	Building a simple C program	304
	Building a simple Pascal program	304
	Building a simple FORTRAN program	304
	11.3.3 Loading the tools	304
	11.3.4 Re-running the tools without reloading them	304
	11.3.5 Running transputer bootable files as MS-DOS commands	305
	11.3.6 The run-time libraries	305
	11.3.7 Transputer memory allocation	306
	The occam memory allocation map	306
	The scientific-language memory allocation map	307
11.3.8	Implementation details	307

	The run-time stack	307
	The run-time heap	308
	Selecting the run-time stack	308
	Placement of the code	309
	The static data area	309
	The scientific-language process communications interface	309
11.3.9	Scientific-language channel i/o support	310
	C support	310
	Pascal support	312
	FORTRAN support	313
	Parallel C support	314
	Parallel FORTRAN support	314
11.3.10	Additional support from Parallel C and Parallel FORTRAN	315
11.3.11	Transputer assembler inserts	316
	Usage of assembler	316
	Local workspace allocation	316
	Review of how the transputer implements procedure calls	317
	The C assembler restrictions and capabilities	318
11.3.12	Mixing OCCAM and non-OCCAM compilation units within the same process	319
	Parameter type compatibilities	319
	Hidden parameters	319
	Array parameters	319
	Vectorspace	320
	OCCAM parameter supersets	320
	Calling an OCCAM FUNCTION	320
11.4	The INMOS D705B occam-2 toolset	321
11.4.1	Software development using the D705B	321
11.4.2	File naming convention	322
11.4.3	Processor types	323
11.4.4	Error modes	324
11.4.5	The makefile generator	325
11.4.6	The occam compiler	325
11.4.7	The syntax checker	326
11.4.8	The librarian	326
11.4.9	The linker	327
11.4.10	Binary lister	327
11.4.11	The bootstrap tool	327
11.4.12	The configurer	327
11.4.13	The debugger	328
11.4.14	The simulator	328
11.4.15	Supplementary tools	328
11.5	Handling non-OCCAM processes	328
11.5.1	Equivalent OCCAM process technology	328
	The Type 1 interface	329
	The Type 2 interface	329
	The Type 3 interface	330
11.5.2	D705B Processor classes	331
11.5.3	EOP Startup and shutdown overheads	331
11.5.4	Practical considerations for writing harnesses	332
	Memory allocation by the standard scientific-language harness	332
	Writing harnesses to allocate scientific-language workspace memory	333
	Placing all EOP stacks below the code	335

	Establishing EOP workspace requirements	335
	Terminating the host file server	337
	Re-running the application without reloading	337
	Process priorities	338
11.6	D705B debugging guidelines	339
	11.6.1 Problems with conventional debugging techniques	339
	11.6.2 Error mode considerations	340
	11.6.3 Run-time debugging aids	340
	11.6.4 Debugging processes that are not connected to the host server	341
	Overview of technique	341
	Implementation detail	341
	What to do if you don't have a debugger	345
11.7	Using the D705B OCCam-2 toolset	345
	11.7.1 About makefiles	345
	11.7.2 Two communicating EOPs on one transputer	345
	Operations overview	346
	The root EOP	346
	The remote EOP	348
	The OCCam bits	349
	Running the program	351
	Rebuilding	352
	Re-implementation of the EOPs	352
	11.7.3 Two communicating EOPs on two transputers	354
	11.7.4 Using the debugger with the twin EOP twin transputer system	356
	11.7.5 Placing the EOPs in a library	356
	11.7.6 Sharing code amongst EOPs in a system	357
	The EOPs	357
	The shared OCCam code	357
	Linker symbol optimization	358
	Calculating where specific modules are placed	359
	Using on-chip RAM effectively	360
	11.7.7 Hints and tips	361
	Library usage guidelines	361
	General usage guidelines	362
11.8	Some useful checklists	363
	11.8.1 Setting things up for the D705B	363
	11.8.2 What to do if a multiple EOP system won't run (on one transputer)	363
	11.8.3 What to do if a multiple EOP system won't run (on many transputers)	364
	11.8.4 A summary of performance maximization techniques	365
11.9	Summary and Conclusions	366
11.10	References	366
A	Quality and Reliability	367
	A Quality and Reliability	368
B	Cables for Board Products	371
	B Cables for board products	372

Preface

Development tools and system products are important and developing areas of application for INMOS devices. The Development and Systems Databook has been published to provide detailed information on the INMOS product range.

The databook comprises an overview, engineering data and applications information for the current range of development tools and systems products.

INMOS provide a wide range of development tools including compilers, toolsets and development kits. A diverse range of software is also available. INMOS systems products provide powerful development platforms for system designers interested in high density, high performance, design simplicity and cost effectiveness.

In addition to development tools and systems products, the INMOS product range also includes transputer products, graphics devices, Digital Signal Processing (DSP) devices and fast SRAMS. For further information concerning INMOS products please contact your local sales outlet.



Systems Products overview

1.1 Introduction

INMOS is a recognised leader in the development and design of high-performance integrated circuits and is a pioneer in the field of parallel processing. The company manufactures components designed to satisfy the most demanding of current processing applications and also provide an upgrade path for future applications. Current designs and development will meet the requirements of systems in the next decade. Computing requirements essentially include high-performance, flexibility and simplicity of use. These characteristics are central to the design of all INMOS products.

INMOS has a consistent record of innovation over a wide product range and, together with its parent company SGS-THOMSON Microelectronics, supplies components to system manufacturing companies in the United States, Europe, Japan and the Far East. INMOS products include a range of transputer products in addition to a highly successful range of high-performance graphics devices, an innovative and successful range of high-performance digital signal processing (DSP) devices and a broad range of fast static RAMs, an area in which it has achieved a greater than 10% market share.

The corporate headquarters, product design team and worldwide sales and marketing management are based at Bristol, UK.

INMOS is constantly upgrading, improving and developing its product range and is committed to maintaining a global position of innovation and leadership.

The Transputer Development and *iq* Systems Databook has been published to assist in the choice of transputer development support products available for SUN, VAX and PC users and to give detailed information on the range of INMOS system products specifically designed for integration into end-user systems. The latter being part of INMOS's *iq* systems business which is dedicated to supplying and servicing the systems builder with innovative and high quality modular products.

Design engineers will find it convenient to use this book in conjunction with The Transputer Databook, one of a series of databooks which specifically detail the INMOS product group areas.

The INMOS transputer family of microprocessors is the industry standard in the field of multi-processing. The family consists of a range of powerful VLSI devices which all adhere to the same basic architecture incorporating a processor, memory and communication links for direct connection to other transputers.

Multiprocessor systems can be constructed from a collection of transputers operating concurrently and communicating through links. Unlike all other microprocessor implementations currently commercially available, additional bus arbitration logic is not required.

1.2 Innovation and Quality

INMOS provides a wide range of tools to support development on the transputer. These have been designed to enable users to easily evaluate transputers and develop systems smoothly within the shortest possible timescales. Development tools include C, FORTRAN and PASCAL compilers and development packages based on the OCCAM compiler. A wide range of software is also available from third parties, details of which are included in The Transputer White Pages Directory, available from INMOS. INMOS provides technical field support, software training courses and a comprehensive software support service.

INMOS has further exploited the power of the transputer architecture and technology by providing a range of modular hardware products for integration into end-user systems and for use as development platforms for general transputer projects. These TRAMs (TRANsputer Modules) are part of a total INMOS strategy to fully support the systems builder in terms of *innovation and quality* for INMOS products and service.

Technical expertise in design and manufacturing of transputer silicon and associated software provides an excellent basis for a professional system product range. INMOS has been a successful supplier of silicon products to the electronics industry for many years and fully recognises the importance of service and quality in the high technology business sectors. In a demanding and competitive marketplace INMOS is fully aware of the critical importance to the system builder of reliable products and effective supplier support.

The experience, expertise and innovation of INMOS combined with the full support and resources of its parent multinational company, SGS-THOMSON Microelectronics, results in a stable yet efficient business foundation.

1.3 TRAMS (TRANsputer Modules)

The INMOS TRAM concept was introduced during 1987 to exploit some of the major benefits of the transputer and parallel processing.

TRAMS are small, cost effective sub-assemblies of transputers and other circuitry (often RAM) with a very simple but efficient 16 signal interface standard profiled in modular sizes. The interface accommodates 4 serial INMOS links for interprocessor communication, power supply and system signals.

With this standard, TRAMs may be mounted onto a variety of motherboards which provide specific host interface hardware. Each motherboard can connect to a number of TRAMs and provides facilities for configuring a network of TRAMs to the user specified topology under software control. A software package is provided for motherboards which allows this task to be undertaken with the minimum of effort.

The TRAM architecture offers many advantages over conventional system configurations. The following features are included:

- An industry standard yet simple multiprocessor interface
- Upgradeability at incremental cost
- Maximum flexibility of cost/performance with minimal real-estate

1.3.1 Standard Interface

The electrical interface to every TRAM product consists of 16 signal pins meeting a standard electrical and mechanical format. All TRAMs are based upon a single module profile with a defined pin layout. This single format is known as Size 1. Larger TRAMs are simply multiple sizes of this format with the same pin spacing. This published format will be maintained for future TRAMs yet to be developed by INMOS and has already been adopted by many third party developers to extend the range of TRAM options and hosts.

1.3.2 Upgradeability

Upgradeability has been a major attraction for customers both for development and end product applications. TRAMs are becoming known as a futureproof solution. Investment made today using TRAMs, with respect to software and hardware engineering, can be regarded as an investment for future designs. Due to the transputers unique ability to distribute application software by booting networks via the links, TRAMs permit exploitation of current hardware technology, but at the same time liberate software development from hardware constraints of cost, real-time performance and compute power. For example, a system designed today by the system integrator may be very efficiently enhanced at some time in the future as a result of a change in his end market demand. This can be achieved in two ways:

- By replacing existing TRAMs with faster transputer silicon TRAMs as they become available (eg, T800-25 to T800-30)
- By adding additional TRAMs to the existing hardware

In either case the system cost is incremental and it is possible to operate with the same original application software by simply reconfiguring and booting the new network. Using traditional sequential multiprocessor solutions the second approach would inevitably result in a complete system hardware and software re-design, significant expansion in board area and a drawn out time to market.

1.3.3 Flexibility

Many system designers exploit the modularity of TRAMs to provide a range of products meeting varieties of performance/cost demand mix. For example, due to the modularity of the hardware and software, a customer may develop a low budget product and a high performance product from the same range of components.

In addition, the same design can be used across the INMOS range of motherboards or specific customer designed motherboards which conform to the TRAM specification. This results in a single design being able to exploit a wide range of host environments and markets.

Unlike other architectural implementations, the preceding flexibility can be achieved utilising just one application software package. A further advantage of this approach is the commonality of TRAM components within each end product type. This offers the system builder significant savings by minimising inventory holding.

1.3.4 Evaluation

Customers investing in the TRAM architecture for transputer evaluation purposes have the opportunity to immediately investigate the performance and characteristics of new transputer silicon as it becomes available. INMOS is committed to provide a standard TRAM interface for each new member of the transputer family.

1.4 Quality and Reliability

All INMOS systems products are manufactured and tested to strict quality standards. Every product undergoes extensive soak testing at temperature before final test. This procedure includes the completion of test logs which are documented and retained for reference.

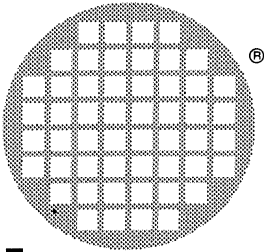
More detailed information describing Quality and Reliability is included later in this publication.



Boards



TRAnspuTer Modules (TRAMS)



inmos

IMS B416 TRAM

16-bit transputer

64 Kbytes

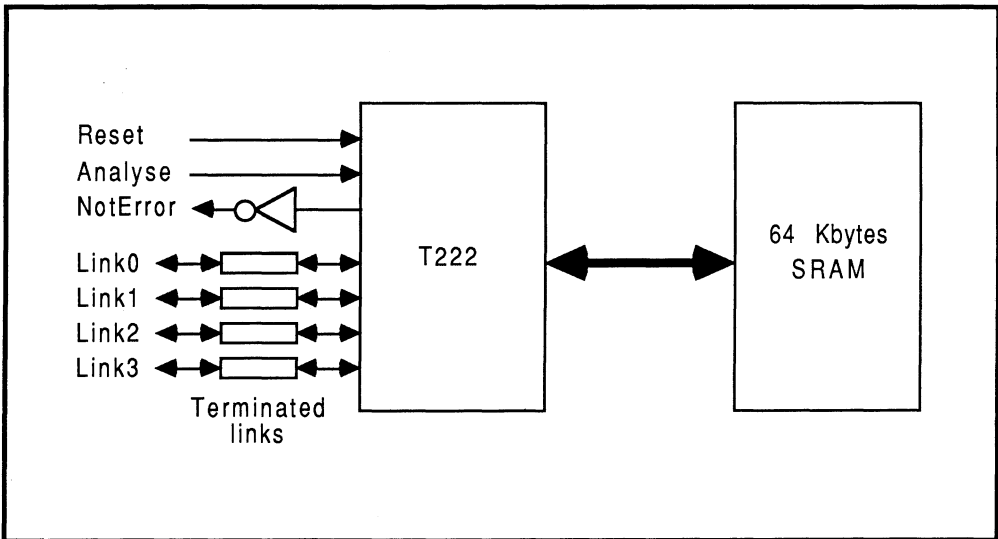
Size 1

FEATURES

- IMS T222 Transputer
- 64 Kbytes of no-wait-state SRAM (100 ns memory cycle time)
- Communicates via 4 INMOS serial links (Selectable between 10 or 20 Mbits/s)
- Package has only 16 active pins.
- Conforms to the TRAM specification

GENERAL DESCRIPTION

The IMS B416 utilises the full memory space of the IMS T222 transputer. It is manufactured fully from surface mount silicon components. The IMS T222's PLCC package brings low cost benefits to TRAM users.



2.1 IMS B416 TRAM engineering data

2.1.1 Introduction

The IMS B416 is one of a range of INMOS TRANputer Modules (TRAMs) incorporating a transputer and 64 Kbytes of static RAM. In effect, these TRAMs are board level transputers with a simple, standardised interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort.

Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Module Motherboard Architecture* and *Dual-In-Line Transputer Modules (TRAMs)* which are included in Part 3 of this databook.

If the user intends to design a custom motherboard, then the *Transputer Databook* will also be required. This is available as a separate publication from INMOS (72 TRN 203 01).

2.1.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC,GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkSpeedA,B	in	Transputer link speed selection	6,7

Table 2.1 IMS B416 Pin designations

Notes:

- 1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- 2 Details of the physical pin locations can be found in Fig. 2.3.

2.1.3 Standard TRAM signals

A TRAM can be regarded as a transputer with extra RAM attached, but with only 16 signals brought out to the TRAM pins. The majority of the TRAM pins function in exactly the same way as the corresponding transputer signals, which are detailed in the *Transputer Databook*. However, a few of these signals are slightly different from the transputer specification as follows:

notError (pin 11)

This is an open collector signal. It is driven low when there is an error; otherwise it is pulled high by a resistor on the motherboard. This enables the **notError** outputs on several TRAMs to be wire-ORed together. (The TRAM specification recommends that no more than 10 **notError** outputs are connected together).

LinkSpeedA and LinkSpeedB (pins 6 and 7)

LinkSpeedA and **LinkSpeedB** set the speed of transputer link 0 and links 1-3 respectively. When the appropriate input is low, the link(s) operate at 10 Mbits/s and when high the links operate at 20 Mbits/s.

Link signals

Whilst the links obey a protocol identical to that described in the *Transputer Databook*, there are some differences in the electrical characteristics.

LinkIn0-3 The link inputs have pull-down resistors to ensure that they are disabled when they are not connected. Diodes are also included for protection against electrostatic discharge.

LinkOut0-3 The link outputs have resistors connected in series for matching to a 100 ohm transmission line.

2.1.4 Memory configuration

The IMS B416 has internal RAM occupying the first 4 Kbytes of address space. The next 60 Kbytes is occupied by the external static RAM present on the board. The total of 64 Kbytes represents the maximum addressable memory space of the IMS T222 transputer.

Table 2.2 details the start and end addresses of this external memory and Fig 2.1 shows a graphical representation of the memory map (the “#” sign indicates a hexadecimal number).

	Hardware byte address
From:	#9000
To:	#7FFE

Table 2.2 Location of external memory on the IMS B416

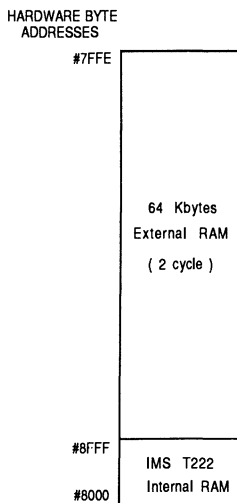


Figure 2.1 Memory map

2.1.5 Mechanical details

Figure 2.2 indicates the vertical dimensions of both a single IMS B416 TRAM and an IMS B416 TRAM stacked on top of another TRAM, and Figure 2.3 shows the outline drawing the of the IMS B416.

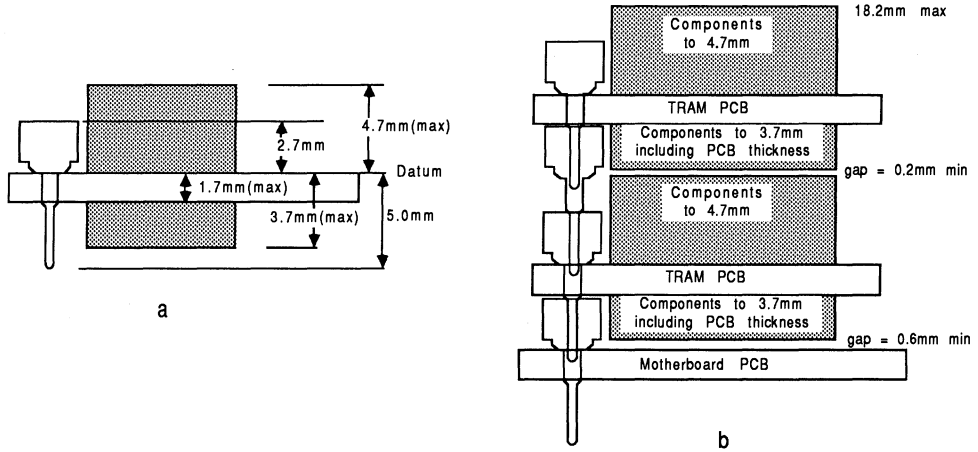


Figure 2.2 IMS B416 height specification, (a) Single TRAM; (b) Two stacked TRAMs

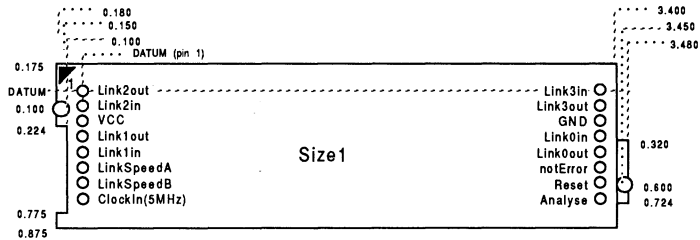


Figure 2.3 IMS B416 outline drawing (all dimensions in inches)

2.1.6 Installation

Since the IMS B416 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B416 is supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

Plug the IMS B416 carefully into the motherboard. Where the IMS B416 is being used with an INMOS motherboard, the silk screened triangle marking pin 1 on the IMS B416 (see Figure 2.3) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot.

Should it be necessary to unplug the IMS B416, it is advised that it is gently levered out while keeping it as flat as possible. As soon as the IMS B416 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

2.1.7 Specification

TRAM feature	IMS B416-10	Unit	Notes
Transputer type	IMS T222-20		
Number of transputers	1		
Number of INMOS serial links	4		
Amount of SRAM	64	Kbytes	
SRAM "wait states"	0		
Amount of DRAM	None		
DRAM "wait states"	N/A		
Memory cycle time	100	ns	
Subsystem controller	No		
Peripheral circuitry	None		
Parity	No		
Size (TRAM size)	1		
Length	3.66	inch	
Pitch between pins	3.30	inch	
Width	1.05	inch	
Component height above PCB	4.7	mm	
Component height below PCB	3.7	mm	1
Weight	–	g	
Storage temperature	0–70	deg C	
Operating temperature	10–40	deg C	2
Power supply voltage (VCC)	4.75–5.25	Volt	
Power consumption	–	W	3

Table 2.3 IMS B416 specification

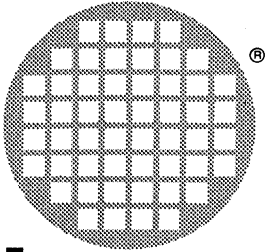
Notes:

- 1 This dimension includes the thickness of the PCB.
- 2 The figure quoted refers to the ambient air temperature.
- 3 The power consumption is the worst case value obtained when a sample of IMS B416 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.

2.1.8 Ordering Information

Description	Order No.
IMS B416 TRAM with IMS T222-20	IMS B416-10

Table 2.4 Ordering information



inmos

IMS B401 TRAM

32-bit transputer

32 Kbytes

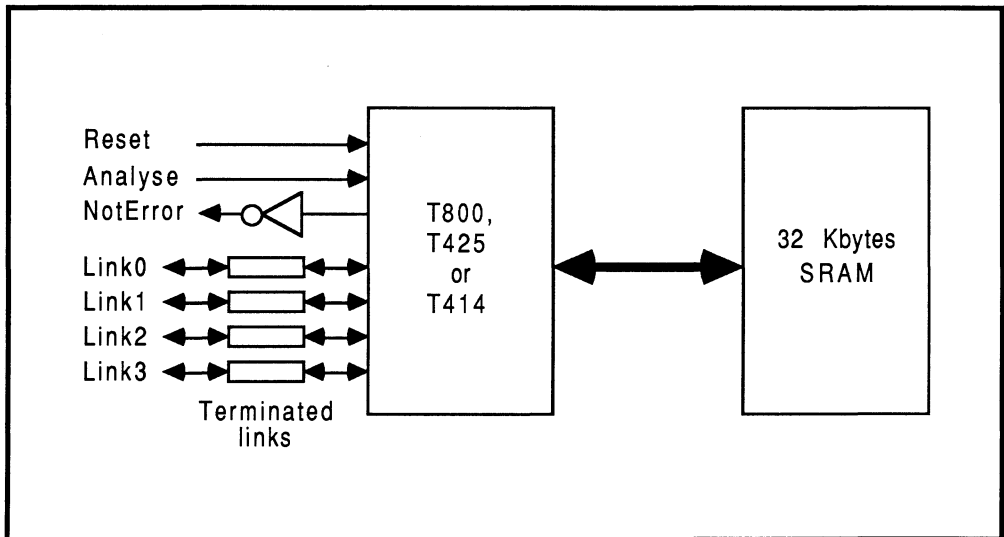
Size 1

FEATURES

- Choice of Transputer (IMS T414, IMS T425 or IMS T800)
- Choice of Processor Speed (20 or 25 MHz)
- 32 Kbytes of no-wait-state SRAM
- Communicates via 4 INMOS serial links (Selectable between 10 or 20 Mbits/s)
- Package has only 16 active pins.
- Designed to a published specification (*INMOS Technical Note 29*).

GENERAL DESCRIPTION

A low cost, high performance, 16 pin transputer, ideal for applications where 4 Kbytes of on-chip RAM is not quite enough. The 32 Kbytes of off chip RAM is ideal for systolic processing, signal processing, feature extraction etc. The IMS B008, fitted with ten IMS B401-5s, offers 50 MWhetstones/s in a single slot of an IBM PC, XT, AT, PS2 model 30, or clone. In the INMOS ITEM, 160 IMS B401-8s offer 2 GIPS (2000 MIPS) and 5 Mbytes.



2.2 IMS B401 TRAM engineering data

2.2.1 Introduction

The IMS B401 is one of a range of INMOS TRANputer Modules (TRAMs) incorporating a transputer and 32 Kbytes of static RAM. In effect, these TRAMs are board level transputers with a simple, standardised interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort.

Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Module Motherboard Architecture* and *Dual-In-Line Transputer Modules (TRAMs)* which are included in Part 3 of this databook.

If the user intends to design a custom motherboard, then *The Transputer Databook* will also be required. This is available as a separate publication from INMOS (72 TRN 203 01).

2.2.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkspeedA,B	in	Transputer link speed selection	6,7

Table 2.1 IMS B401 Pin designations

Notes:

- 1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- 2 Details of the physical pin locations can be found in Fig. 2.3.

2.2.3 Standard TRAM signals

A TRAM can be regarded as a transputer with extra RAM attached, but with only 16 signals brought out to the TRAM pins. The majority of the TRAM pins function in exactly the same way as the corresponding transputer signals, which are detailed in *The Transputer Databook*. However, a few of these signals are slightly different from the transputer specification as follows:

notError (pin 11)

This is an open collector signal. It is driven low when there is an error; otherwise it is pulled high by a resistor on the motherboard. This enables the **notError** outputs on several TRAMs to be wire-ORed together. (The TRAM specification recommends that no more than 10 **notError** outputs are connected together).

LinkSpeedA and LinkSpeedB (pins 6 and 7)

LinkspeedA and **LinkspeedB** set the speed of transputer link 0 and links 1-3 respectively. When the appropriate input is low the link(s) operate at 10 Mbits/s and when high the link(s) operate at 20 Mbits/s.

Link signals

Whilst the links obey a protocol identical to that described in *The Transputer Databook*, there are some differences in the electrical characteristics.

LinkIn0-3 The link inputs have pull-down resistors to ensure that they are disabled when they are not connected. Diodes are also included for protection against electrostatic discharge.

LinkOut0-3 The link outputs have resistors connected in series for matching to a 100 ohm transmission line.

2.2.4 Memory configuration

The IMS B401 with a IMS T414 has internal RAM occupying the first 2 Kbytes of address space, whereas internal RAM occupies the first 4 Kbytes on the IMS B401 with an IMS 425 or IMS T800. The two memory maps shown in Figure 2.1 reflect this fact.

Location of external memory

The IMS B401 has 32 Kbytes of external memory. Tables 2.2 and 2.3 show the start addresses of this memory for both the IMS T414 and the IMS T425/T800 versions of the IMS B401 (the “#” sign indicates a hexadecimal number).

	Hardware byte address
From:	#80000800
To:	#800087FF

Table 2.2 Location of external memory on the IMS B401 with IMS T414

	Hardware byte address
From:	#80001000
To:	#80008FFF

Table 2.3 Location of external memory on the IMS B401 with IMS T425 or IMS T800

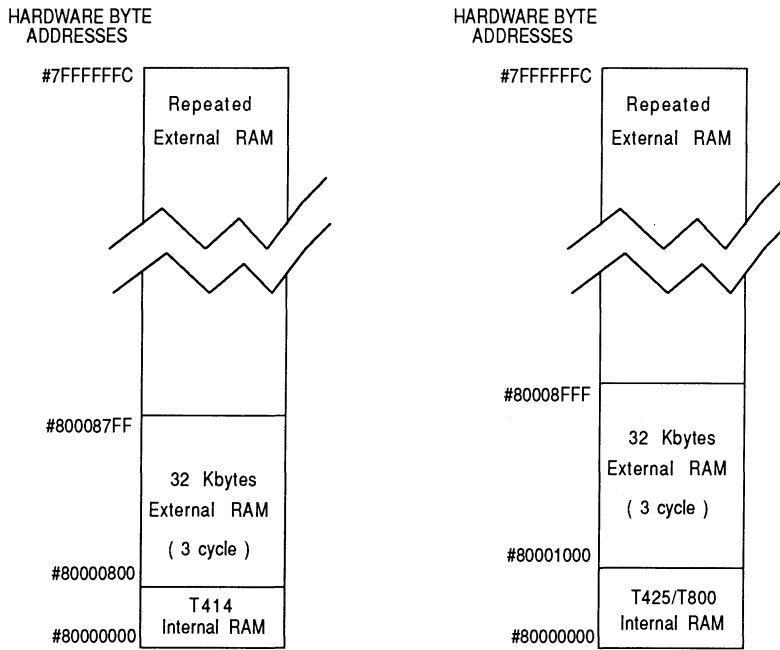


Figure 2.1 Memory maps

2.2.5 Mechanical details

Figure 2.2 indicates the vertical dimensions of both a single IMS B401 TRAM and an IMS B401 TRAM stacked on top of another TRAM, and Figure 2.3 shows the outline drawing the of the IMS B401.

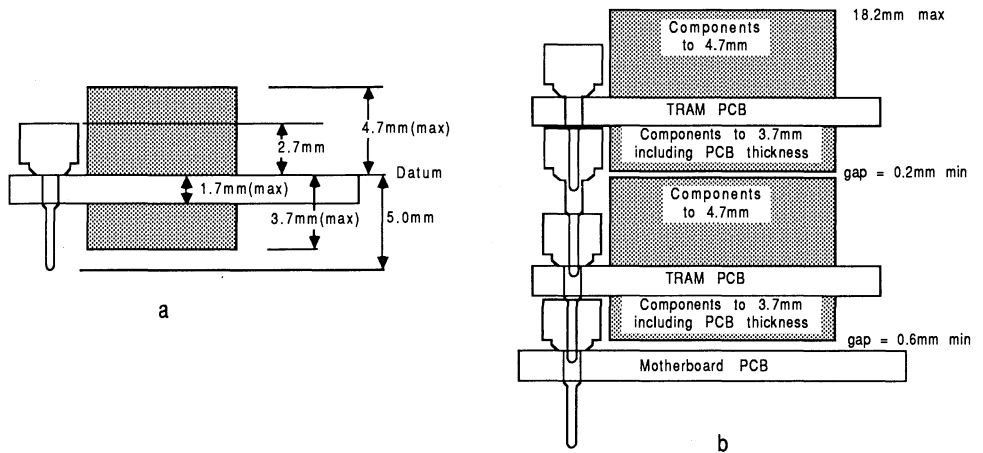


Figure 2.2 IMS B401 height specification, (a) Single TRAM; (b) Two stacked TRAMs

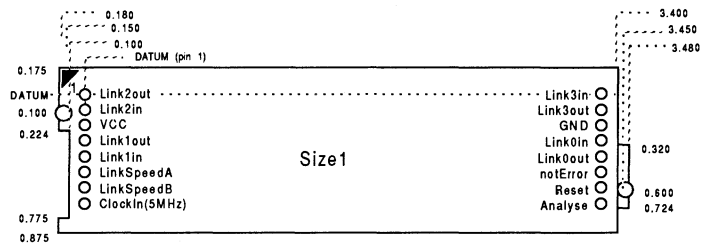


Figure 2.3 IMS B401 outline drawing (all dimensions in inches)

2.2.6 Installation

Since the IMS B401 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B401 is supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

Plug the IMS B401 carefully into the motherboard. Where the IMS B401 is being used with an INMOS motherboard, the silk screened triangle marking pin 1 on the IMS B401 (see Figure 2.3) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot. If it is envisaged that the assembly is likely to be subjected to any vibrations, it is recommended that the TRAM is secured to the motherboard using nylon M3 nuts and bolts. The bolts should be inserted through the fixing holes on the motherboard, and through the castellations on two edges of the TRAM. A number of these nuts and bolts are supplied with each of the INMOS motherboards.

Should it be necessary to unplug the IMS B401, it is advised that, having removed any retaining nuts and bolts, it is gently levered out while keeping it as flat as possible. As soon as the IMS B401 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

2.2.7 Specification

TRAM feature	IMS B401-2	IMS B401-3	IMS B401-8	IMS B401-5	Unit	Notes
Transputer type	T414-20	T800-20	T425-25	T800-25		
Number of transputers	1	1	1	1		
Number of INMOS serial links	4	4	4	4		
Amount of SRAM	32	32	32	32	Kbytes	
SRAM "wait states"	0	0	0	0		
SRAM cycle time	150	150	120	120	ns	
Amount of DRAM	None	None	None	None		
DRAM "wait states"	N/A	N/A	N/A	N/A		
DRAM cycle time	N/A	N/A	N/A	N/A		
Subsystem controller	No	No	No	No		
Peripheral circuitry	None	None	None	None		
Parity	No	No	No	No		
Size (TRAM size)	1	1	1	1		
Length	3.66	3.66	3.66	3.66	inch	
Pitch between pins	3.30	3.30	3.30	3.30	inch	
Width	1.05	1.05	1.05	1.05	inch	
Component height above PCB	4.7	4.7	4.7	4.7	mm	
Component height below PCB	3.7	3.7	3.7	3.7	mm	1
Weight	21	21	21	21	g	
Storage temperature	0-70	0-70	0-70	0-70	deg C	
Operating temperature	10-40	10-40	10-40	10-40	deg C	2
Power supply voltage (VCC)	4.75-5.25	4.75-5.25	4.75-5.25	4.75-5.25	Volt	
Power consumption	1.2	1.2	1.5	1.5	W	3

Table 2.4 IMS B401 specification

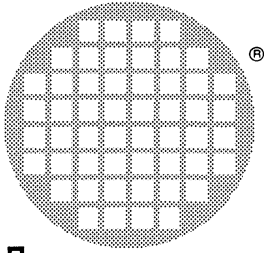
Notes:

- 1 This dimension includes the thickness of the PCB.
- 2 The figure quoted refers to the ambient air temperature.
- 3 The power consumption is the worst case value obtained when a sample of IMS B401 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.

2.2.8 Ordering Information

Description	Order Number
IMS B401 TRAM with IMS T414-20	IMS B401-2
IMS B401 TRAM with IMS T800-20	IMS B401-3
IMS B401 TRAM with IMS T425-25	IMS B401-8
IMS B401 TRAM with IMS T800-25	IMS B401-5

Table 2.5 Ordering information



inmos

IMS B411 TRAM

32-bit transputer

1 Mbytes

Size 1

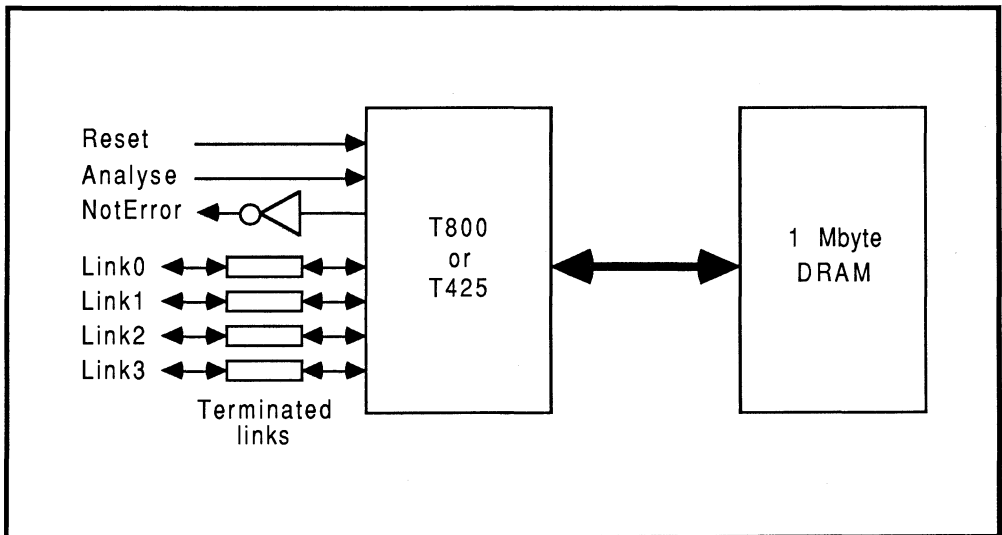
FEATURES

- Choice of IMS T425 or IMS T800 Transputer
- 1 Mbyte of zero wait-state DRAM (150 ns memory cycle time)
- Size 1 TRAM
- Communicates via 4 INMOS serial links
- Package has only 16 active pins.
- Conforms to the TRAM specification

GENERAL DESCRIPTION

The IMS B411 TRAM is the ideal module for applications where space is at a premium. With a full Mbyte of DRAM on a size 1 TRAM, 8 transputers and 8 Mbytes of memory can be installed in a single IBM PC (*) slot (using the IMS B008 motherboard) or in a single 6U VMEbus slot (using the IMS B014 motherboard). The choice of an IMS T800 or T425 transputer gives the user the flexibility to tailor a system to his exact requirements in terms of cost and performance.

(*) For IBM PC read: original PC, XT, AT, PS/2 Model 30 and most clones.



2.3 IMS B411 TRAM engineering data

2.3.1 Description

The IMS B411 is an INMOS TRAnsputer Module (TRAM) incorporating either an IMS T800 or IMS T425 transputer and 1 Mbyte of dynamic RAM.

TRAMs are board level transputers with a simple, standardised interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort. TRAMs may be plugged into motherboards, which provide the necessary electrical signals, mechanical support and usually, an interface to a host machine. Various motherboards are now available from INMOS and from third-party vendors for most of the common computing platforms.

Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Module Motherboard Architecture* and *Dual-In-Line Transputer Modules (TRAMs)* which are included in Part 3 of this databook.

If the user intends to design a custom motherboard, then *The Transputer Databook* will also be required. This is available as a separate publication from INMOS (72 TRN 203 01).

2.3.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkspeedA,B	in	Transputer link speed selection	6,7

Table 2.1 IMS B411 Pin designations

Notes:

- 1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- 2 Details of the physical pin locations can be found in Fig. 2.3.

2.3.3 Standard TRAM signals

A TRAM can be regarded as a transputer with extra RAM attached, but with only 16 signals brought out to the TRAM pins. The majority of the TRAM pins function in exactly the same way as the corresponding transputer signals, which are detailed in *The Transputer Databook*. However, a few of these signals are slightly different from the transputer specification as follows:

notError (pin 11)

This is an open collector signal. It is driven low when there is an error; otherwise it is pulled high by a resistor on the motherboard. This enables the **notError** outputs on several TRAMs to be wire-ORed together. The TRAM specification recommends that no more than 10 **notError** outputs are connected together).

LinkSpeedA and LinkSpeedB (pins 6 and 7)

LinkSpeedA and **LinkSpeedB** set the speed of transputer link 0 and links 1-3 respectively. When the appropriate input is low the link(s) operate at 10 Mbits/s and when high the link(s) operate at 20 Mbits/s.

Link signals

Whilst the links obey a protocol identical to that described in the *Transputer Databook*, there are some differences in the electrical characteristics.

LinkIn0-3 The link inputs have pull-down resistors to ensure that they are disabled when they are not connected. Diodes are also included for protection against electrostatic discharge.

LinkOut0-3 The link outputs have resistors connected in series for matching to a 100 ohm transmission line.

2.3.4 Memory configuration

The internal RAM of the IMS T800 or IMS T425 occupies the first 4 Kbytes of address space. The next 1 Mbyte is occupied by the external dynamic RAM present on the TRAM. The external RAM is repeated in 1 Mbyte blocks throughout the higher address space.

Table 2.2 details the start and end addresses of the external memory and Fig 2.1 shows a graphical representation of the memory map (the “#” sign indicates a hexadecimal number).

	Hardware byte address
From:	#80001000
To:	#80100FFF

Table 2.2 Location of external memory on the IMS B411

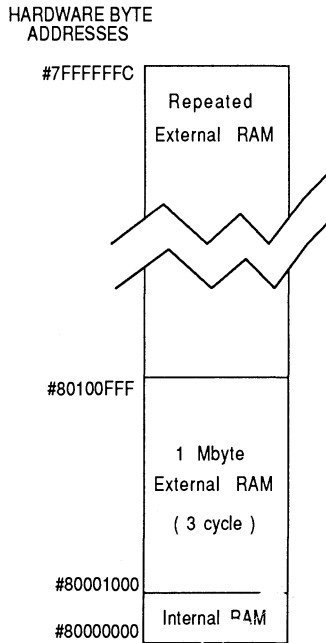


Figure 2.1 Memory map

2.3.5 Mechanical details

Figure 2.2 indicates the vertical dimensions of a single IMS B411 TRAM, and Figure 2.3 shows the outline drawing of the IMS B411.

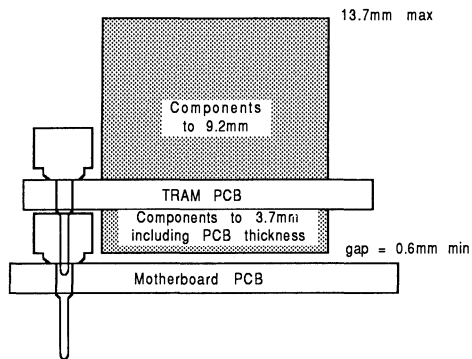


Figure 2.2 IMS B411 height specification

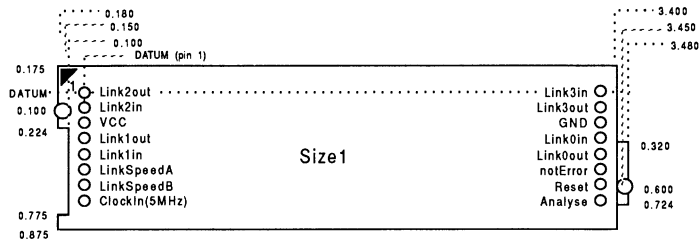


Figure 2.3 IMS B411 outline drawing (all dimensions in inches)

2.3.6 Installation

Since the IMS B411 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B411 is supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

Plug the IMS B411 carefully into the motherboard. Where the IMS B411 is being used with an INMOS motherboard, the silk screened triangle marking pin 1 on the IMS B411 (see Figure 2.3) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot. If it is envisaged that the assembly is likely to be subjected to any vibrations, it is recommended that the TRAM is secured to the motherboard using nylon M3 nuts and bolts. The bolts should be inserted through the fixing holes on the motherboard, and through the castlations on two edges of the TRAM. A number of these nuts and bolts are supplied with each of the INMOS motherboards.

Should it be necessary to unplug the IMS B411, it is advised that, having removed any retaining nuts and bolts, it is gently levered out while keeping it as flat as possible. As soon as the IMS B411 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

2.3.7 Specification

TRAM feature	IMS B411-3	IMS B411-7	Unit	Notes
Transputer type	T800-20	T425-20		
Number of transputers	1	1		
Number of INMOS serial links	4	4		
Amount of SRAM	None	None		
SRAM "wait states"	N/A	N/A		
Amount of DRAM	1	1	Mbyte	
DRAM "wait states"	0	0		
Memory cycle time	150	150	ns	
Subsystem controller	No	No		
Peripheral circuitry	None	None		
Parity	No	No		
Size (TRAM size)	1	1		
Length	3.66	3.66	inch	
Pitch between pins	3.30	3.30	inch	
Width	1.05	1.05	inch	
Component height above PCB	9.2	9.2	mm	
Component height below PCB	3.7	3.7	mm	1
Weight	50	50	g	
Storage temperature	0-70	0-70	deg C	
Operating temperature	10-40	10-40	deg C	2
Power supply voltage (VCC)	4.75-5.25	4.75-5.25	Volt	
Power consumption	4	4	W	3

Table 2.3 IMS B411 specification

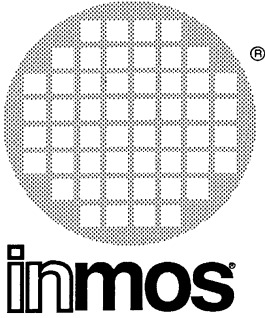
Notes:

- 1 This dimension includes the thickness of the PCB.
- 2 The figure quoted refers to the ambient air temperature.
- 3 The power consumption is the worst case value obtained when a sample of IMS B411 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.

2.3.8 Ordering Information

Description	Order Number
IMS B411 TRAM with IMS T800-20	IMS B411-3
IMS B411 TRAM with IMS T425-20	IMS B411-7

Table 2.4 Ordering information



IMS B404 TRAM

32-bit transputer

2 Mbytes

Size 2

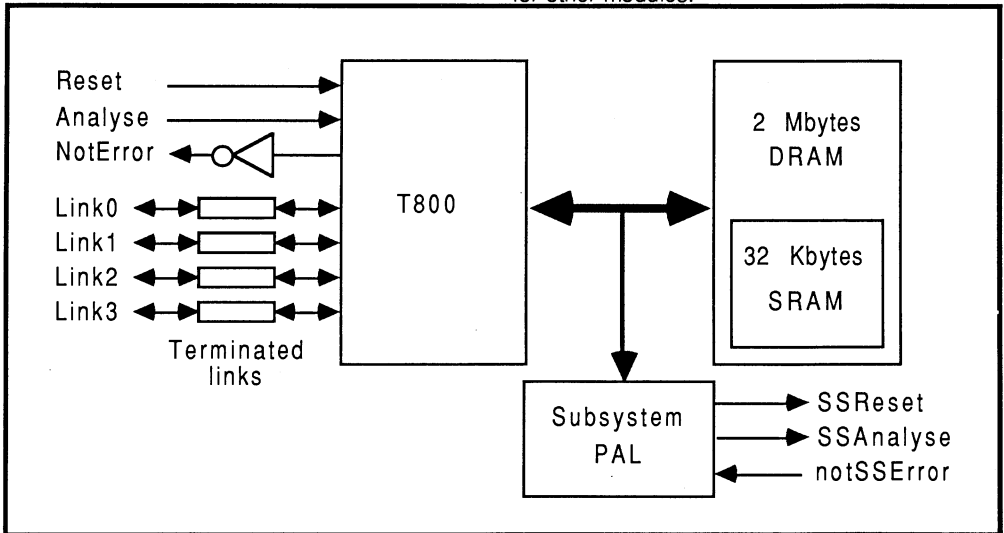
FEATURES

- IMS T800 Transputer
- 32 Kbytes of zero wait-state SRAM
- 2 Mbytes of single wait-state DRAM
- Subsystem controller circuitry
- Communicates via 4 INMOS serial links (Selectable between 10 or 20 Mbits/s)
- Package has only 16 active pins
- Designed to a published specification

GENERAL DESCRIPTION

The IMS B404 is a very compact compute module providing a full 2 Mbytes of memory and still providing maximum performance capability. This is achieved by extending the principle of fast on chip RAM to include 32 Kbytes of static RAM which cycles as fast as possible. So any technique which puts most frequently accessed memory locations near the bottom of memory will speed up the processing. This TRAM is the most popular board for running INMOS' TDS or Toolset packages.

The IMS B404 packs 11 cm² of silicon onto a board the size of a credit card. Four IMS B404s fit onto the IMS B008 in a single slot of the IBM PC. Fifty IMS B404s fit into an ITEM, to give 100 Mbytes, 625 MIPS, 250 MWhetstones, with space to spare for other modules.



2.4 IMS B404 TRAM engineering data

2.4.1 Introduction

The IMS B404 is one of a range of INMOS **TRAN**puter **Modules** (TRAMs) incorporating a IMS T800 transputer, 32 Kbytes of static RAM and 2 Mbytes of dynamic RAM. In effect, these TRAMs are board level transputers with a simple, standardized interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort.

Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Module Motherboard Architecture* and *Dual-In-Line Transputer Modules (TRAMs)* which are included in Part 3 of this databook.

If the user intends to design a custom motherboard, then *The Transputer Databook* will also be required. This is available as a separate publication from INMOS (72 TRN 203 01).

2.4.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkspeedA,B	in	Transputer link speed selection	6,7
Subsystem Services			
SubSystemReset	out	Subsystem reset	1b
SubSystemAnalyse	out	Subsystem error analysis	1c
notSubSystemError	in	Subsystem error indicator	1a

Table 2.1 IMS B404 Pin designations

Notes:

- 1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- 2 Details of the physical pin locations can be found in Fig. 2.3.

2.4.3 Standard TRAM signals

A TRAM can be regarded as a transputer with extra RAM attached, but with only 16 signals brought out to the TRAM pins. The majority of the TRAM pins function in exactly the same way as the corresponding transputer signals, which are detailed in *The Transputer Databook*. However, a few of these signals are slightly different from the transputer specification as follows:

notError (pin 11)

This is an open collector signal. It is driven low when there is an error; otherwise it is pulled high by a resistor on the motherboard. This enables the **notError** outputs on several TRAMs to be wire-ORed together. (The TRAM specification recommends that no more than 10 **notError** outputs are connected together).

LinkSpeedA and LinkSpeedB (pins 6 and 7)

LinkSpeedA and **LinkSpeedB** set the speed of transputer link 0 and links 1-3 respectively. When the appropriate input is low, the link(s) operate at 10 Mbits/s, and when high the link(s) operate at 20 Mbits/s.

Link signals

Whilst the links obey a protocol identical to that described in the *Transputer Databook*, there are some differences in the electrical characteristics.

LinkIn0-3 The link inputs have pull-down resistors to ensure that they are disabled when they are not connected. Diodes are also included for protection against electrostatic discharge.

LinkOut0-3 The link outputs have resistors connected in series for matching to a 100 ohm transmission line.

2.4.4 Subsystem signals

The IMS B404 has a subsystem port in addition to the usual TRAM signals. This enables the TRAM to reset or analyse a subsystem of other TRAMs and/or motherboards. The polarity of these signals is the same as that of the **Reset**, **Analyse** and **notError** standard TRAM signals. Therefore, the IMS B404 subsystem can drive other TRAMs on the same motherboard with no intermediate logic. However, **SubSystemReset** and **SubSystemAnalyse** must go through inverting buffers if they are to drive a subsystem off the motherboard.

These subsystem signals are accessed by writing or reading to control registers in the transputer memory space. See Section 2.4.5.

2.4.5 Memory configuration

The IMS B404 is able to access 2 Mbytes of memory. This is comprised of 4 Kbytes of internal transputer memory, 28 Kbytes of external SRAM and 2016 Kbytes of external DRAM. There are, in fact, 32 Kbytes of SRAM components and 2 Mbytes of DRAM components on the board, but the address spaces of each type of memory are superimposed. Therefore, the total memory available is limited to 2 Mbytes. This is sufficient to enable the Transputer Development System (TDS) to be run on a single IMS B404 TRAM.

Location of external memory

Tables 2.2 and 2.3 show the start addresses of the different types of external memory on the IMS B404 (the “#” sign indicates a hexadecimal number). The internal RAM on the IMS T800 occupies the first 4 Kbytes of address space.

	Hardware byte address
From:	#80001000
To:	#80007FFF

Table 2.2 Location of external SRAM on the IMS B404

	Hardware byte address
From:	#80008000
To:	#801FFFFF

Table 2.3 Location of external DRAM on the IMS B404

Since the internal memory on the IMS T800 is 1 cycle, the external SRAM is 3 cycle and the DRAM is 4 (or 5) cycle, a memory speed hierarchy is established. This architecture allows programmers to structure their code for optimum performance, and will become of even greater significance when the next faster version of the transputer becomes available.

Subsystem register locations

The subsystem register addresses start at hardware address #00000000 in all TRAMs that utilize a 32-bit processor, allowing software compatibility between TRAMs. These registers are located as shown in Table 2.4.

Register	Hardware byte address
SubSystemReset (write only)	#00000000
SubSystemAnalyse (write only)	#00000004
notSubSystemError (read only)	#00000000

Table 2.4 Subsystem address locations

Setting bit 0 in either the reset or the analyse registers asserts the corresponding signal. Similarly, clearing bit 0 deasserts the signal. When an error occurs in the subsystem, bit 0 of the error location becomes set.

Byte locations #00000008 and #0000000C are unused. The subsystem registers are repeated at every sixteenth byte location in the positive address space. See Figure 2.1.

HARDWARE BYTE
ADDRESSES

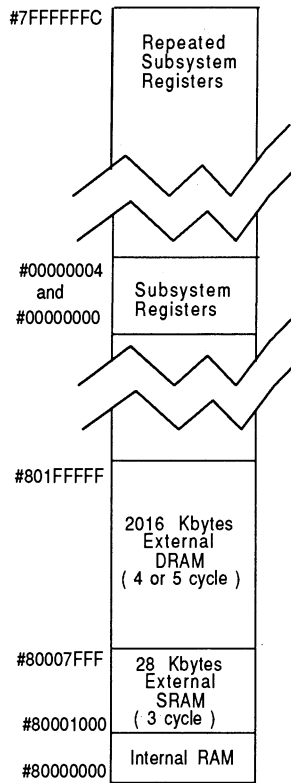


Figure 2.1 Memory map

2.4.6 Mechanical details

Figure 2.2 indicates the vertical dimensions of a single IMS B404 and Figure 2.3 shows the outline drawing of the IMS B404.

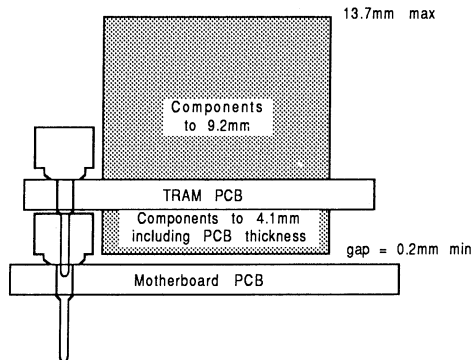


Figure 2.2 IMS B404 height specification

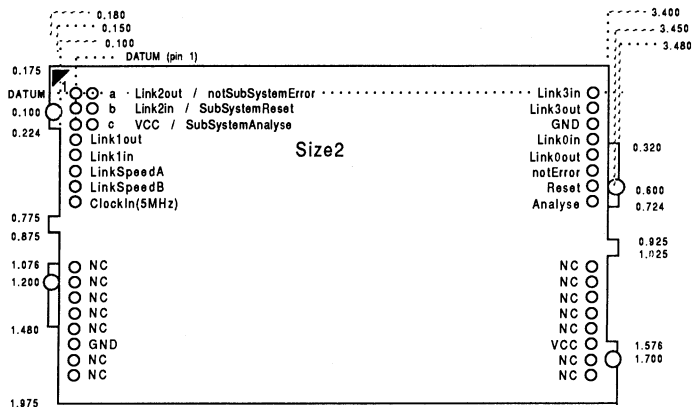


Figure 2.3 IMS B404 outline drawing (All dimensions in inches)

2.4.7 Installation

Since the IMS B404 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B404 is supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

If the subsystem signals are required, plug a 3-way header strip into the solder-side sockets on the IMS B404.

Plug the IMS B404 into the motherboard. Where the IMS B404 is being used with an INMOS motherboard, the silk screened triangle marking pin 1 on the IMS B404 (see Figure 2.3) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot.

Should it be necessary to unplug the IMS B404, it is advised that it is gently levered out while keeping it as flat as possible. As soon as the IMS B404 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

2.4.8 Specification

This specification applies to part numbers IMS B404-5 x , IMS B404-3 y and IMS B404-4 z , where x denotes Rev. A and after, y denotes Rev. E and after, and z denotes Rev. C and after.

TRAM feature	IMS B404-5	IMS B404-3	IMS B404-4	Unit	Notes
Transputer type	IMS T800-25	IMS T800-20	IMS T800-17		
Number of transputers	1	1	1		
Number of INMOS serial links	4	4	4		
Amount of SRAM	32	32	32	Kbyte	
SRAM "wait states"	0	0	0		
Amount of DRAM	2	2	2	Mbyte	
DRAM "wait states"	1(2)	1	1		6
Memory cycle time	120/160(200)	150/200	171/229	ns	1,6
Subsystem controller	Yes	Yes	Yes		
Peripheral circuitry	None	None	None		
Parity	No	No	No		
Size (TRAM size)	2	2	2		
Length	3.66	3.66	3.66	inch	
Pitch between pins	3.30	3.30	3.30	inch	
Width	2.15	2.15	2.15	inch	
Component height above PCB	9.2	9.2	9.2	mm	2
Component height below PCB	3.7	3.7	3.7	mm	3
Weight	68	68	68	g	
Storage temperature	0–70	0–70	0–70	deg C	
Operating temperature	10–40	10–40	10–40	deg C	4
Power supply voltage (VCC)	4.75–5.25	4.75–5.25	4.75–5.25	Volt	
Power consumption	6	5	5	W	5

Table 2.5 IMS B404 specification

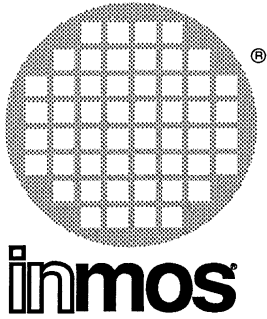
Notes:

- 1 The two figures quoted refer to (i) SRAM cycle time, and (ii) DRAM cycle time.
- 2 Since the IMS B404 makes use of 1 Mbit ZIP RAMs, this dimension is larger than is normally stated for TRAMs.
- 3 This dimension includes the thickness of the PCB.
- 4 The figure quoted refers to the ambient air temperature.
- 5 The power consumption is the worst case value obtained when a sample of IMS B404 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.
- 6 The B404-5 incorporates some wait-state logic that generates an extra memory cycle wait-state whenever the same bank of DRAM is accessed consecutively. The figures in parentheses refer to these instances.

2.4.9 Ordering Information

Description	Order Number
IMS B404 TRAM with IMS T800-25	IMS B404-5
IMS B404 TRAM with IMS T800-20	IMS B404-3
IMS B404 TRAM with IMS T800-17	IMS B404-4

Table 2.6 Ordering information



IMS B417 TRAM

32-bit transputer

4 Mbytes

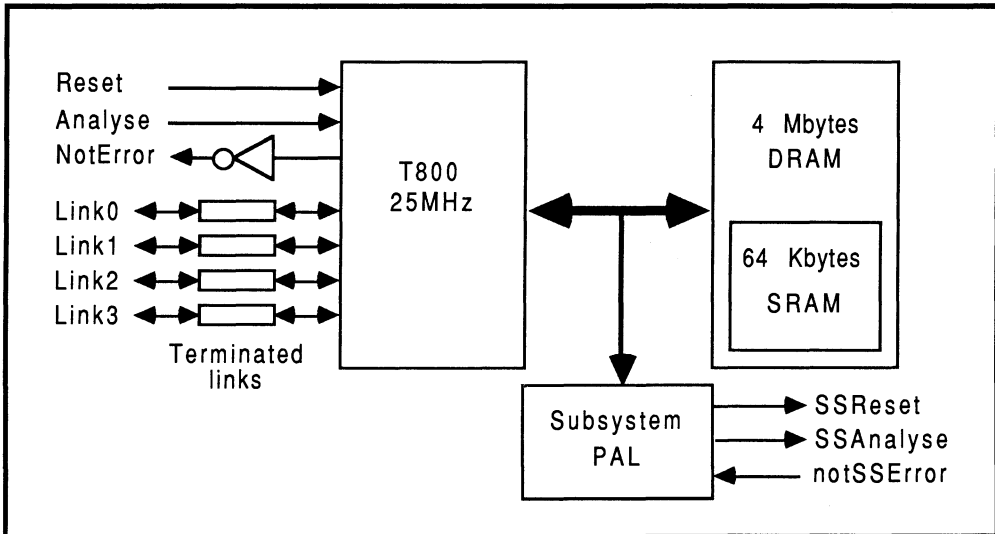
Size 4

FEATURES

- IMS T800 25 MHz Transputer
- 64 Kbytes of zero wait-state SRAM
- 4 Mbytes of single wait-state DRAM
- Subsystem controller circuitry
- Communicates via 4 INMOS serial links (Selectable between 10 or 20 Mbits/s)
- Package has only 16 active pins
- Designed to a published specification

GENERAL DESCRIPTION

The IMS B417 uses the IMS T800 25MHz transputer (the highest speed T800). The 4 Mbytes of DRAM is sufficient to run the Ada compiler from Alsys. Also provided is 64 Kbytes of fast SRAM (3 cycle), so any technique which puts most frequently accessed memory locations near the bottom of memory will speed up the processing.



2.5 IMS B417 TRAM engineering data

2.5.1 Introduction

The IMS B417 is one of a range of INMOS **TRAnspuTer Modules** (TRAMs) incorporating a IMS T800 transputer, 64 Kbytes of static RAM and 4 Mbytes of dynamic RAM. In effect, these TRAMs are board level transputers with a simple, standardized interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort.

Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Module Motherboard Architecture* and *Dual-In-Line Transputer Modules (TRAMs)* which are included in Part 3 of this databook.

If the user intends to design a custom motherboard, then *The Transputer Databook* will also be required. This is available as a separate publication from INMOS (72 TRN 203 01).

2.5.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkspeedA,B	in	Transputer link speed selection	6,7
Subsystem Services			
SubSystemReset	out	Subsystem reset	1b
SubSystemAnalyse	out	Subsystem error analysis	1c
notSubSystemError	in	Subsystem error indicator	1a

Table 2.1 IMS B417 Pin designations

Notes:

- 1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- 2 Details of the physical pin locations can be found in Fig. 2.3.

2.5.3 Standard TRAM signals

A TRAM can be regarded as a transputer with extra RAM attached, but with only 16 signals brought out to the TRAM pins. The majority of the TRAM pins function in exactly the same way as the corresponding transputer signals, which are detailed in *The Transputer Databook*. However, a few of these signals are slightly different from the transputer specification as follows:

notError (pin 11)

This is an open collector signal. It is driven low when there is an error; otherwise it is pulled high by a resistor on the motherboard. This enables the **notError** outputs on several TRAMs to be wire-ORed together. (The TRAM specification recommends that no more than 10 **notError** outputs are connected together).

LinkSpeedA and LinkSpeedB (pins 6 and 7)

LinkSpeedA and **LinkSpeedB** set the speed of transputer link 0 and links 1-3 respectively. When the appropriate input is low, the link(s) operate at 10 Mbits/s, and when high the link(s) operate at 20 Mbits/s.

Link signals

Whilst the links obey a protocol identical to that described in *The Transputer Databook*, there are some differences in the electrical characteristics.

LinkIn0-3 The link inputs have pull-down resistors to ensure that they are disabled when they are not connected. Diodes are also included for protection against electrostatic discharge.

LinkOut0-3 The link outputs have resistors connected in series for matching to a 100 ohm transmission line.

2.5.4 Subsystem signals

The IMS B417 has a subsystem port in addition to the usual TRAM signals. This enables the TRAM to reset or analyse a subsystem of other TRAMs and/or motherboards. The polarity of these signals is the same as that of the **Reset**, **Analyse** and **notError** standard TRAM signals. Therefore, the IMS B417 subsystem can drive other TRAMs on the same motherboard with no intermediate logic. However, **SubSystemReset** and **SubSystemAnalyse** must go through inverting buffers if they are to drive a subsystem off the motherboard.

These subsystem signals are accessed by writing or reading to control registers in the transputer memory space. See Section 2.5.5.

2.5.5 Memory configuration

The IMS B417 is able to access 4 Mbytes of memory. This is comprised of 4 Kbytes of internal transputer memory, 60 Kbytes of external SRAM and 4032 Kbytes of external DRAM. There are, in fact, 64 Kbytes of SRAM components and 4 Mbytes of DRAM components on the board, but the address spaces of each type of memory are superimposed. Therefore, the total memory available is limited to 4 Mbytes.

Location of external memory

Tables 2.2 and 2.3 show the start addresses of the different types of external memory on the IMS B417 (the “#” sign indicates a hexadecimal number). The internal RAM on the IMS T800 occupies the first 4 Kbytes of address space.

	Hardware byte address
From:	#80001000
To:	#8000FFFF

Table 2.2 Location of external SRAM on the IMS B417

	Hardware byte address
From:	#80010000
To:	#803FFFFF

Table 2.3 Location of external DRAM on the IMS B417

Since the internal memory on the IMS T800 is 1 cycle, the external SRAM is 3 cycle and the DRAM is 4 cycle, a memory speed hierarchy is established. This architecture allows programmers to structure their code for optimum performance.

Subsystem register locations

The subsystem register addresses start at hardware address #00000000 in all TRAMs that utilize a 32-bit processor, allowing software compatibility between TRAMs. These registers are located as shown in Table 2.4.

Register	Hardware byte address
SubSystemReset (write only)	#00000000
SubSystemAnalyse (write only)	#00000004
notSubSystemError (read only)	#00000000

Table 2.4 Subsystem address locations

Setting bit 0 in either the reset or the analyse registers asserts the corresponding signal. Similarly, clearing bit 0 deasserts the signal. When an error occurs in the subsystem, bit 0 of the error location becomes set.

Byte locations #00000008 and #0000000C are unused. The subsystem registers are repeated at every sixteenth byte location in the positive address space. See Figure 2.1.

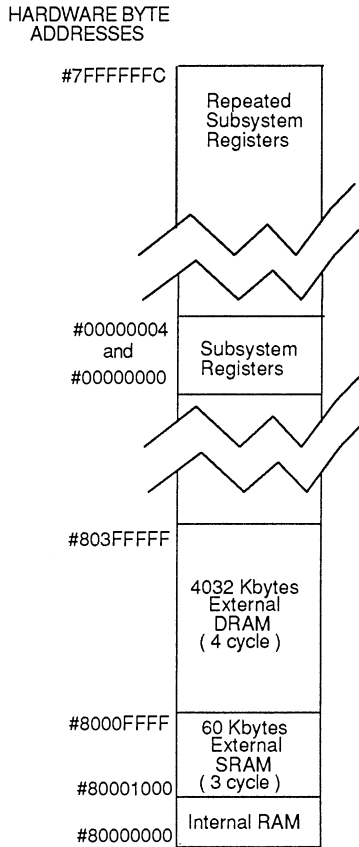


Figure 2.1 Memory map

2.5.6 Mechanical details

Figure 2.2 indicates the vertical dimensions of a single IMS B417 and Figure 2.3 shows the outline drawing of the IMS B417.

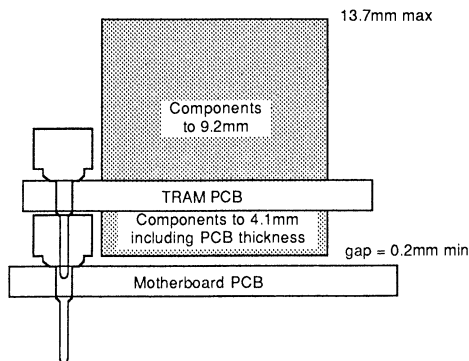


Figure 2.2 IMS B417 height specification

2.5.8 Specification

TRAM feature	IMS B417-5	Unit	Notes
Transputer type	IMS T800-25		
Number of transputers	1		
Number of INMOS serial links	4		
Amount of SRAM	64	Kbyte	
SRAM "wait states"	0		
SRAM cycle time	120	ns	
Amount of DRAM	4	Mbyte	
DRAM "wait states"	1		
DRAM cycle time	160	ns	
Subsystem controller	Yes		
Peripheral circuitry	None		
Parity	No		
Size (TRAM size)	4		
Length	3.66	inch	
Pitch between pins	3.30	inch	
Width	4.35	inch	
Component height above PCB	9.2	mm	
Component height below PCB	3.7	mm	1
Weight	–	g	
Storage temperature	0–70	deg C	
Operating temperature	10–40	deg C	2
Power supply voltage (VCC)	4.75–5.25	Volt	
Power consumption	–	W	3

Table 2.5 IMS B417 specification

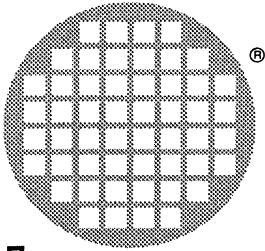
Notes:

- 1 This dimension includes the thickness of the PCB.
- 2 The figure quoted refers to the ambient air temperature.
- 3 The power consumption is the worst case value obtained when a sample of IMS B417 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.

2.5.9 Ordering Information

Description	Order Number
IMS B417 TRAM with IMS T800-25	IMS B417-5

Table 2.6 Ordering information



inmos

IMS B405 TRAM

32-bit transputer

8 Mbytes

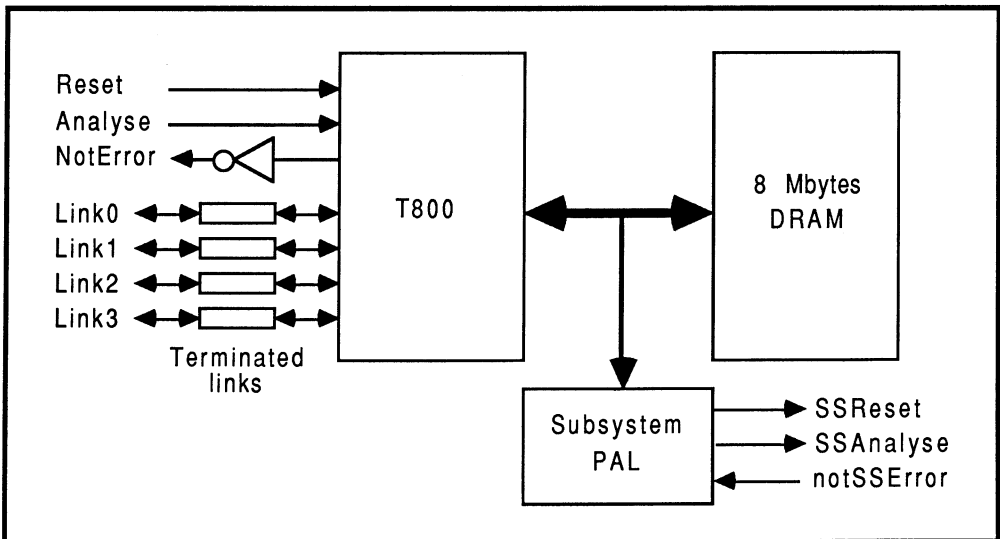
Size 8

FEATURES

- IMS T800 Transputer
- 8 Megabytes of DRAM
- User selectable byte parity checking
- Subsystem controller circuitry
- Stackability allows other TRAMs to be "piggy-backed"
- Package has only 16 active pins
- Designed to a published specification

GENERAL DESCRIPTION

The IMS B405 puts 8 Mbytes, with parity, onto a module that fits comfortably on a IMS B008 in the IBM PC, considerably more volume efficient than a 20 Mbyte Winchester disk. Although the IMS B405 is marginally slower than the IMS B403 and IMS B404, its large amount of external RAM makes it an ideal board for running existing large programs, such as CAD, AI, simulation, etc.



2.6 IMS B405 TRAM engineering data

2.6.1 Introduction

The IMS B405 is one of a range of INMOS TRANsputer Modules (TRAMs) incorporating a transputer and 8 Mbytes of dynamic RAM. In effect, these TRAMs are board level transputers with a simple, standardised interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort.

Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Module Motherboard Architecture* and *Dual-In-Line Transputer Modules (TRAMs)* which are included in Part 3 of this databook.

If TRAMs are to be designed into a system using a custom motherboard, then *The Transputer Databook* will also be required. This is available as a separate publication from INMOS.

2.6.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkspeedA,B	in	Transputer link speed selection	6,7
Subsystem Services			
SubSystemReset	out	Subsystem reset	1b
SubSystemAnalyse	out	Subsystem error analysis	1c
notSubSystemError	in	Subsystem error indicator	1a

Table 2.1 IMS B405 Pin designations

Notes:

- 1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- 2 Details of the physical pin locations can be found in Fig. 2.3.

2.6.3 Standard TRAM signals

A TRAM can be regarded as a transputer with extra RAM attached, but with only 16 signals brought out to the TRAM pins. The majority of the TRAM pins function in exactly the same way as the corresponding transputer signals, which are detailed in *The Transputer Databook*. However, a few of these signals are slightly different from the transputer specification as follows:

notError (pin 11)

This is an open collector signal. It is driven low when there is an error; otherwise it is pulled high by a resistor on the motherboard. This enables the **notError** outputs on several TRAMs to be wire-ORed together. (The *Dual-In-Line Transputer Modules (TRAMs)* document recommends that no more than 10 **notError** outputs are connected together).

LinkSpeedA and LinkSpeedB (pins 6 and 7)

LinkSpeedA and **LinkSpeedB** set the speed of transputer link 0 and links 1-3 respectively. When the appropriate input is low, the link(s) operate at 10 Mbits/s, and when high the link(s) operate at 20 Mbits/s.

Link signals

Whilst the links obey a protocol identical to that described in *The Transputer Databook*, there are some differences in the electrical characteristics.

LinkIn0-3 The link inputs have pull-down resistors to ensure that they are disabled when they are not connected. Diodes are also included for protection against electrostatic discharge.

LinkOut0-3 The link outputs have resistors connected in series for matching to a 100 ohm transmission line.

2.6.4 Subsystem signals

The IMS B405 has a subsystem port in addition to the usual TRAM signals. This enables the TRAM to reset or analyse a subsystem of other TRAMs and/or motherboards. The polarity of these signals is the same as that of the **Reset**, **Analyse** and **notError** standard TRAM signals. Therefore, the IMS B405 subsystem can drive other TRAMs on the same motherboard with no intermediate logic. However, **SubSystemReset** and **SubSystemAnalyse** must go through inverting buffers if they are to drive a subsystem off the motherboard.

These subsystem signals are accessed by writing or reading to control registers in the transputer memory space. See Section 2.6.5.

2.6.5 Memory configuration

The IMS B405 has internal RAM occupying the first 4 Kbytes of address space, followed by 8 Mbytes of external RAM.

Location of external memory

Table 2.2 shows the location of the external memory on the IMS B405 TRAM (the “#” sign indicates a hexadecimal number).

	Hardware byte address
From:	#80001000
To:	#80800000

Table 2.2 Location of external memory on the IMS B405

Subsystem register locations

The subsystem register addresses start at hardware address #00000000 in all TRAMs with 32-bit processors, allowing software compatibility between TRAMs. These registers are located as shown in Table 2.3.

Register	Hardware byte address
SubSystemReset (write only)	#00000000
SubSystemAnalyse (write only)	#00000004
notSubSystemError (read only)	#00000000

Table 2.3 Subsystem address locations

Setting bit 0 in either the reset or the analyse registers asserts the corresponding signal. Similarly, clearing bit 0 deasserts the signal. When an error occurs in the subsystem, bit 0 of the error location becomes set.

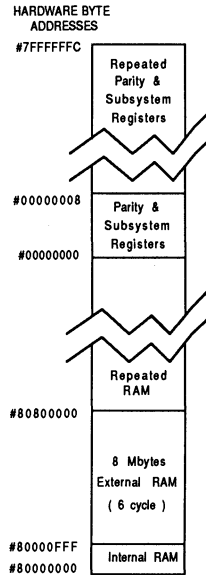


Figure 2.1 Memory map

Memory parity

The IMS B405 includes parity logic for external RAM. This may be enabled/disabled by the user. The aim behind the inclusion of this circuitry on the IMS B405 is to ensure that there is no way that corrupt data can reach any other transputer in the system.

If a parity error occurs, the wait signal is held active so that no more memory cycles are completed. Although all data is lost if a parity error occurs, this is much preferable to corrupt data being further processed by the system.

Parity is enabled or disabled by writing to a parity control register (see Table 2.4). If parity is enabled and an error occurs, it is indicated by **notError** being asserted. The information regarding the cause of the error can then be established by interrogating the parity status register.

Reset disables parity checking and deasserts **MemWait**. **Analyse** causes **MemWait** to be deasserted, while the contents of the parity control register are preserved.

Register	Hardware byte address
Parity control register (write only)	#00000008
Parity status register (read only)	#00000008

Table 2.4 Parity register locations

Setting bit 0 of the parity control register enables parity error detection. Similarly, clearing bit 0 disables parity error detection. When a parity error occurs bit 0 in the parity status register becomes set. Bits 1 & 2 then indicate the BYTE in which the error has occurred (bit 1 is LSB), and bit 3 indicates the BANK in which the error has occurred.

2.6.6 Installation

Since the IMS B405 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B405 is supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

If the subsystem signals are required, plug a 3-way header strip into the solder-side sockets on the IMS B405.

Plug the IMS B405 into the motherboard. Where the IMS B405 is being used with an INMOS motherboard, the silk screened triangle marking pin 1 on the IMS B405 (see Figure 2.3) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot.

Should it be necessary to unplug the IMS B405, it is advised that it is gently levered out while keeping it as flat as possible. As soon as the IMS B405 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

2.6.7 Mechanical details

Figure 2.2 indicates the vertical dimensions of both a single IMS B405 and two IMS B405 TRAMs stacked on top of each other, and Figure 2.3 shows the outline drawing of the IMS B405.

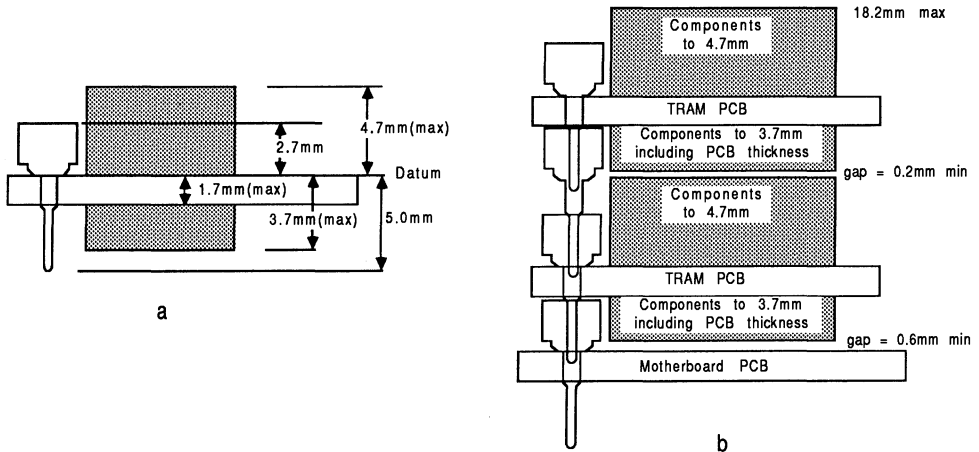


Figure 2.2 IMS B405 height specification, (a) Single TRAM; (b) Two stacked TRAMs

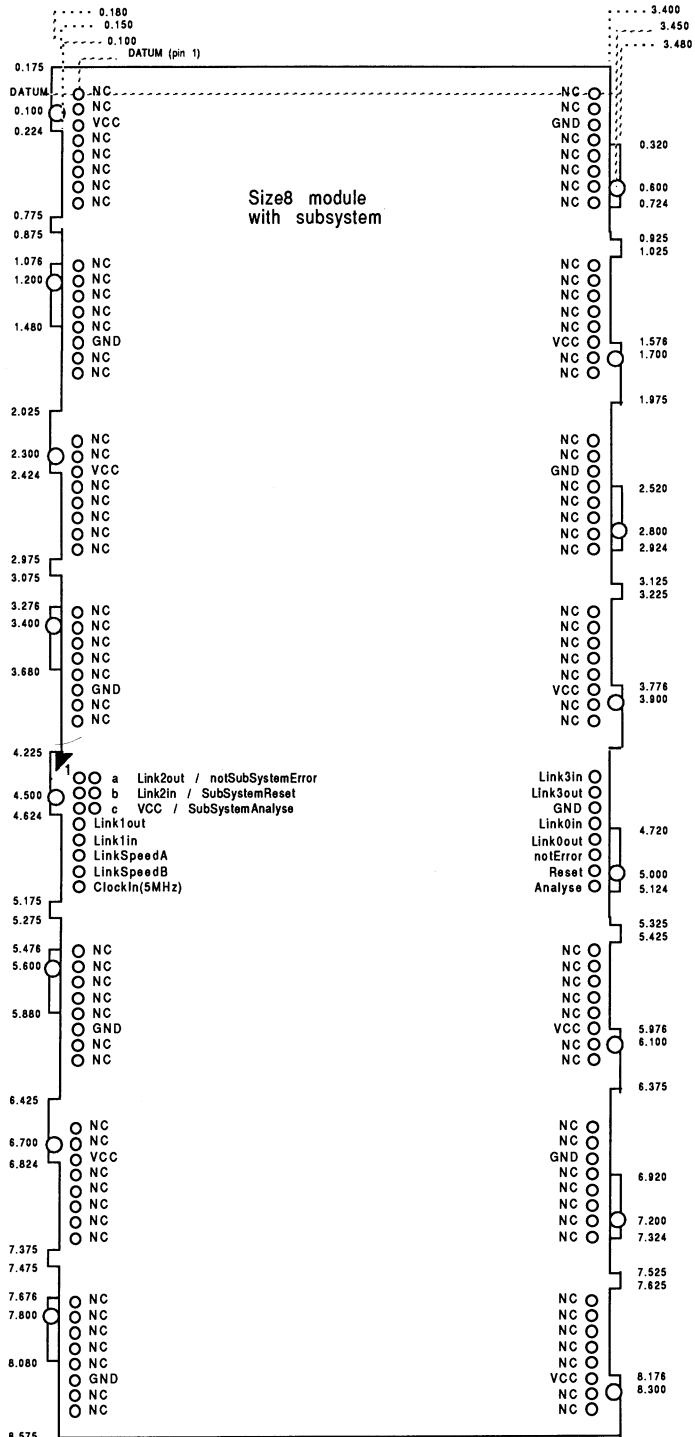


Figure 2.3 IMS B405 outline drawing (All dimensions in inches)

2.6.8 Specification

TRAM feature		Unit	Notes
Transputer type	T800-20		
Number of transputers	1		
Number of INMOS serial links	4		
Amount of SRAM	None		
SRAM "wait states"	N/A		
Amount of DRAM	8	Mbyte	
DRAM "wait states"	2		1
Memory cycle time	300	ns	1
Subsystem controller	Yes		
Peripheral circuitry	None		
Parity	Yes		
Size (TRAM size)	8		
Length	3.66	inch	
Pitch between pins	3.30	inch	
Width	8.75	inch	
Component height above PCB	4.7	mm	
Component height below PCB	3.7	mm	2
Weight	280	g	
Storage temperature	0–70	deg C	
Operating temperature	10–40	deg C	3
Power supply voltage (VCC)	4.75–5.25	Volt	
Power consumption	12	W	4

Table 2.5 IMS B405 specification

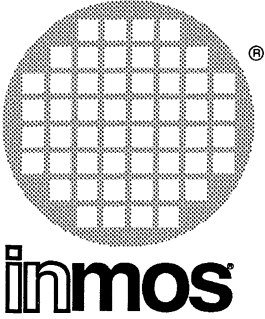
Notes:

- 1 Parity enabled.
- 2 This dimension includes the thickness of the PCB.
- 3 The figure quoted refers to the ambient air temperature.
- 4 The power consumption is the worst case value obtained when a sample of IMS B405 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.

2.6.9 Ordering Information

Description	Order Number
IMS B405 TRAM with IMS T800-20	IMS B405-3

Table 2.6 Ordering information



IMS B410 TRAM

32-bit transputer

160 Kbytes

Size 2

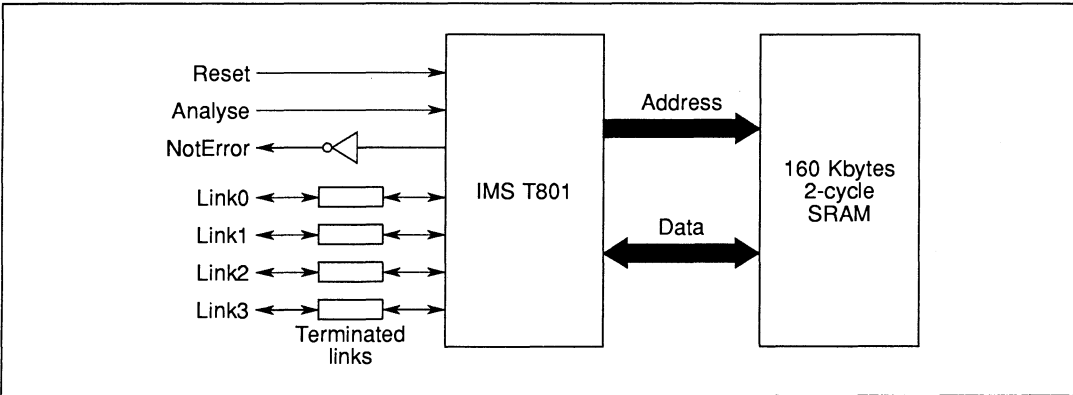
FEATURES

- IMS T801 Transputer with demultiplexed address and data busses
- 160 Kbytes of no-wait-state SRAM (100 ns memory cycle time)
- Size 2 TRAM
- Communicates via 4 INMOS serial links
- Stackability allows other TRAMs to be "piggy-backed"
- Package has only 16 active pins.
- Conforms to the TRAM specification

GENERAL DESCRIPTION

The de-multiplexed address and data busses of the IMS T801 transputer allow very high performance systems to be constructed. The IMS B410 TRAM achieves 2-cycle memory accesses with very fast SRAMs, and yet still manages to squeeze 160 Kbytes onto a size 2 TRAM.

The IMS B410 TRAM allows users to benchmark the performance of the IMS T801 transputer. The standard TRAM interface means that the processor can simply be plugged into existing development systems. However, this module is as equally at home in very high performance system products, as it is in the evaluation environment.



2.7 IMS B410 TRAM engineering data

2.7.1 Description

The IMS B410 is an INMOS Transputer Module (TRAM) incorporating the IMS T801 transputer and 160 Kbytes of static RAM. The demultiplexed address and data busses of the IMS T801 allow maximum use to be made of some very fast static RAM, and the IMS B410 achieves 2-cycle, 100ns external memory cycles.

TRAMs are board level transputers with a simple, standardised interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort. TRAMs may be plugged into motherboards, which provide the necessary electrical signals, mechanical support and usually, an interface to a host machine. Various motherboards are now available from INMOS and from third-party vendors for most of the common computing platforms.

Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Module Motherboard Architecture* and *Dual-In-Line Transputer Modules (TRAMs)* which are included in Part 3 of this databook.

If the user intends to design a custom motherboard, then *The Transputer Databook* will also be required. This is available as a separate publication from INMOS (72 TRN 203 01).

2.7.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkspeedA,B	in	Transputer link speed selection	6,7

Table 2.1 IMS B410 Pin designations

Notes:

- 1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- 2 Details of the physical pin locations can be found in Fig. 2.3.

2.7.3 Standard TRAM signals

A TRAM can be regarded as a transputer with extra RAM attached, but with only 16 signals brought out to the TRAM pins. The majority of the TRAM pins function in exactly the same way as the corresponding transputer signals, which are detailed in *The Transputer Databook*. However, a few of these signals are slightly different from the transputer specification as follows:

notError (pin 11)

This is an open collector signal. It is driven low when there is an error; otherwise it is pulled high by a resistor on the motherboard. This enables the **notError** outputs on several TRAMs to be wire-ORed together. The TRAM specification recommends that no more than 10 **notError** outputs are connected together).

LinkSpeedA and LinkSpeedB (pins 6 and 7)

LinkspeedA and **LinkspeedB** set the speed of transputer link 0 and links 1-3 respectively. When the appropriate input is low the link(s) operate at 10 Mbits/s and when high the link(s) operate at 20 Mbits/s.

Link signals

Whilst the links obey a protocol identical to that described in *The Transputer Databook*, there are some differences in the electrical characteristics.

LinkIn0-3 The link inputs have pull-down resistors to ensure that they are disabled when they are not connected. Diodes are also included for protection against electrostatic discharge.

LinkOut0-3 The link outputs have resistors connected in series for matching to a 100 ohm transmission line.

2.7.4 Memory configuration

The internal RAM of the IMS T801 occupies the first 4 Kbytes of address space. The next 156 Kbytes is occupied by the external static RAM present on the TRAM. There is then a gap of 96 Kbytes. A pattern of 160 K of external RAM followed by a gap is repeated in 256 Kbyte blocks throughout the higher address space.

Table 2.2 details the start and end addresses of the external memory and Fig 2.1 shows a graphical representation of the memory map (the “#” sign indicates a hexadecimal number).

	Hardware byte address
From:	#80001000
To:	#80027FFF

Table 2.2 Location of external memory on the IMS B410

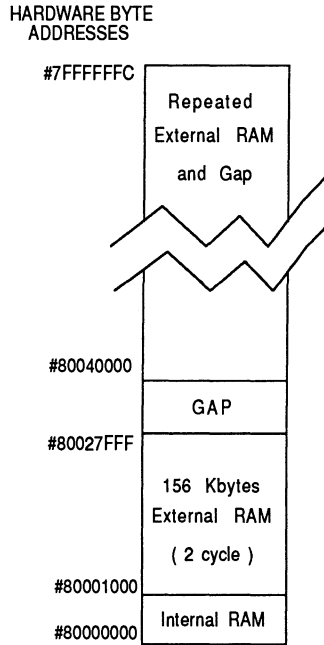


Figure 2.1 Memory map

2.7.5 Mechanical details

Figure 2.2 indicates the vertical dimensions of both a single IMS B410 TRAM and an IMS B410 TRAM stacked on top of another TRAM, and Figure 2.3 shows the outline drawing of the IMS B410.

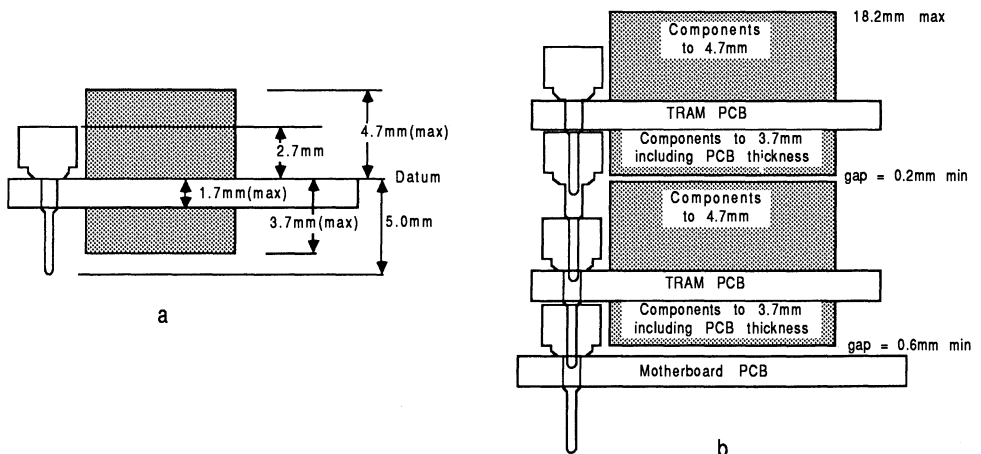


Figure 2.2 IMS B410 height specification, (a) Single TRAM; (b) Two stacked TRAMs

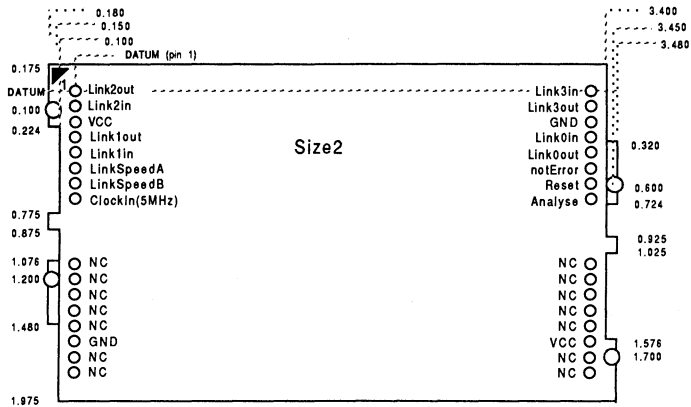


Figure 2.3 IMS B410 outline drawing (all dimensions in inches)

2.7.6 Installation

Since the IMS B410 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B410 is supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

Plug the IMS B410 carefully into the motherboard. Where the IMS B410 is being used with an INMOS motherboard, the silk screened triangle marking pin 1 on the IMS B410 (see Figure 2.3) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot. If it is envisaged that the assembly is likely to be subjected to any vibrations, it is recommended that the TRAM is secured to the motherboard using nylon M3 nuts and bolts. The bolts should be inserted through the fixing holes on the motherboard, and through the castlignations on two edges of the TRAM. A number of these nuts and bolts are supplied with each of the INMOS motherboards.

Should it be necessary to unplug the IMS B410, it is advised that, having removed any retaining nuts and bolts, it is gently levered out while keeping it as flat as possible. As soon as the IMS B410 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

2.7.7 Specification

TRAM feature	IMS B410-11	Unit	Notes
Transputer type	T801-20		
Number of transputers	1		
Number of INMOS serial links	4		
Amount of SRAM	160	Kbytes	
SRAM "wait states"	0		
Amount of DRAM	None		
DRAM "wait states"	N/A		
Memory cycle time	100	ns	
Subsystem controller	No		
Peripheral circuitry	None		
Parity	No		
Size (TRAM size)	2		
Length	3.66	inch	
Pitch between pins	3.30	inch	
Width	2.15	inch	
Component height above PCB	4.7	mm	
Component height below PCB	3.7	mm	1
Weight	50	g	
Storage temperature	0–70	deg C	
Operating temperature	10–40	deg C	2
Power supply voltage (VCC)	4.75–5.25	Volt	
Power consumption	3	W	3

Table 2.3 IMS B410 specification

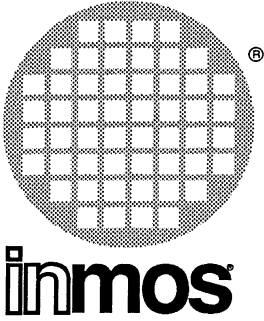
Notes:

- 1 This dimension includes the thickness of the PCB.
- 2 The figure quoted refers to the ambient air temperature.
- 3 The power consumption is the worst case value obtained when a sample of IMS B410 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.

2.7.8 Ordering Information

Description	Order Number
IMS B410 TRAM with IMS T801-20	IMS B410-11

Table 2.4 Ordering information



IMS B415

Differential link buffer TRAM

Size 2

FEATURES

- Buffers all TRAM signals to RS422-compatible differential drive
- Handles 4 links reset and subsystem services signals
- Capable of 20 Mbits/s link operation
- Links go quiet when disconnected
- Designed for 100 ohm twisted pair cable
- $\pm 7V$ common-mode noise rejection
- Size 2 TRAM

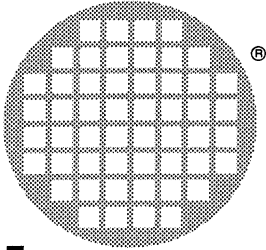
GENERAL DESCRIPTION

The IMS B415 differential interface buffer TRAM allows connections between transputer systems which are not in the same electrical environment. No common ground connection is required, reducing earthing problems. With cable lengths up to 10m, 20 Mbit/s link speed is possible. Longer cables up to 100m support lower link speeds.

Ordering Information

Description	Order Number
IMS B415 Differential link TRAM	IMS B415-1

Table 1: Ordering information



inmos

IMS B418

Flash ROM TRAM

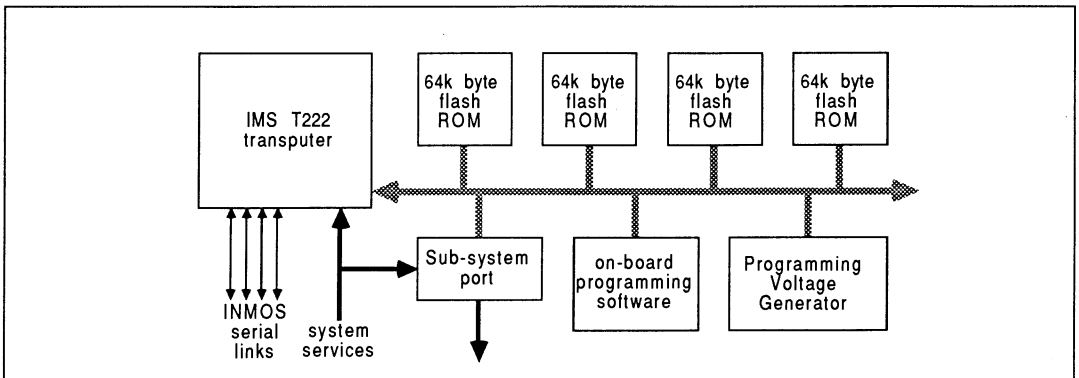
Size 2

FEATURES

- 256 Kbytes non-volatile memory (Flash ROM)
- 10 000 program/erase cycles
- ideal for booting embedded transputer systems
- in-system reprogrammability
- ROM contents user-programmed through INMOS link
- size 2 TRAM
- sub-system port for resetting transputer networks
- can be used as non-volatile backup memory

GENERAL DESCRIPTION

The IMS B418 is a TRAM designed primarily for configuring and boot-strapping transputer networks in embedded systems. It contains 256 Kbytes of non-volatile memory: implemented with *flash ROM* devices (the *flash ROM* is an EPROM-like device with bulk electrical erasability, rather than UV erase). After reset, the IMS B418 outputs a program stored in the ROM from one of its INMOS serial links. An on-board programming voltage generator, and programming software, allows the ROM contents to be programmed without removing the ROM devices from the board and without removing the IMS B418 from an assembled system. Programming is through a simple protocol on one of the INMOS serial links. Safeguards are provided against accidental erasure/programming. The ROM devices can be reprogrammed at least 10 000 times. The IMS B418 can also be used as a non-volatile backup memory in any microprocessor system: all that is required is an INMOS link adaptor to interface the IMS B418 to the microprocessor.



2.9 IMS B418 flash-ROM TRAM product overview

2.9.1 Specification

TRAM feature		Unit	Notes
Flash ROM	256	kbytes	1
TRAM size	2		
Length	3.66	inch	
Width	2.15	inch	
Pitch between pins	3.30	inch	
Component height above PCB	9.0	mm	
Component height below PCB	3.5	mm	2
Weight	60	g	
Storage temperature	0–70	°C	
Operating temperature	10–40	°C	
Power supply voltage (VCC)	4.75–5.25	Volt	
Power consumption	TBA	W	

Table 2.1 IMS B418 specification

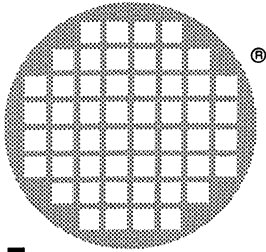
Notes:

- 1 A full 256k bytes are available to the user: the on-board programming software used on the IMS B418 is stored in a separate device.
- 2 This dimension includes the thickness of the PCB.

2.9.2 Ordering Information

Description	Order Number
IMS B418 flash-ROM TRAM	IMS B418-10

Table 2.2 Ordering information



inmos

IMS B407

Ethernet TRAM

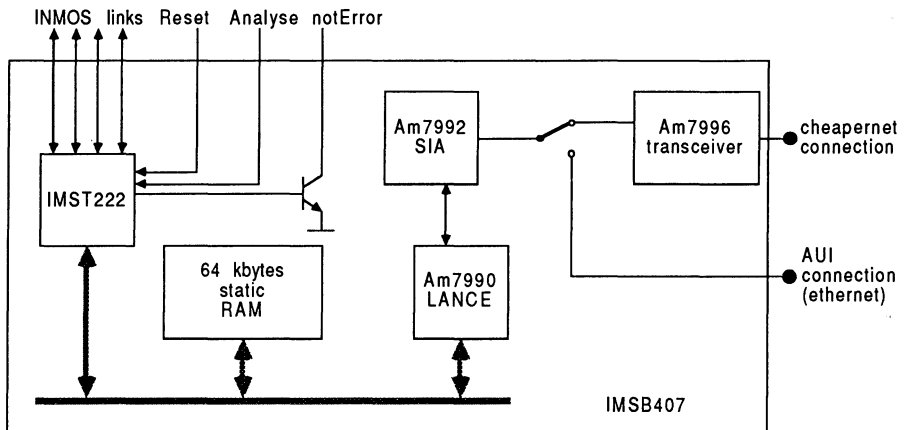
Size 8

FEATURES

- Connects transputer systems to IEEE802.3 Local Area Networks (Ethernet or Cheapernet)
- IMS T222, 16-bit Transputer
- 64 kbytes SRAM, 150ns access cycle
- Uses Am7990 (LANCE) Ethernet Controller
- Communicates via 4 INMOS serial links (Selectable between 10 or 20 Mbits/s)
- Designed to a published specification (*INMOS Technical Note 29*)

GENERAL DESCRIPTION

The IMS B407 is an INMOS link to Ethernet / Cheapernet interface. It allows transputer systems to be connected to other computers and computer networks via IEEE802.3 Local Area Networks (LANs). The IMS B407 can be connected to Ethernet systems (IEEE802.3 10BASE5 or Type A); or to Cheapernet systems (IEEE802.3 10BASE2 or Type B). The IMS B407 consists of an IMS T222 16 bit transputer with 64 kbytes of SRAM. The Ethernet / Cheapernet interface is implemented with the Am 7990 (LANCE), Am 7992 and Am 7996. The Cheapernet interface is isolated to 500V dc. An Attachment Unit Interface (AUI) is provided for connection to Ethernet Media Access Units.



2.10 IMS B407 TRAM engineering data

2.10.1 Transputer Modules (TRAMs)

The IMS B407 is one of a range of INMOS TRANputer Modules (TRAMs). TRAMs are subassemblies of transputers, memory and peripheral devices. They interface to each other via INMOS links, have a standard pinout, and come in a range of standard sizes. ¹ TRAMs allow powerful, flexible, transputer based systems to be produced with the minimum of design effort. The standard TRAM interface signals are described below.

2.10.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkSpeedA,B	in	Transputer link speed selection	6,7

Table 2.1 IMS B407 Pin designations

Notes:

- 1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- 2 Details of the physical pin locations can be found in Fig. 2.8.

LinkOut0-3 Transputer link output signals. These outputs are intended to drive into transmission lines with a characteristic impedance of 100Ω. They can be connected directly to the **LinkIn** pins of other transputers or TRAMs.

LinkIn0-3 Transputer link input signals. These are the link inputs of the transputer on the IMS B407. Each input has a 10kΩ resistor to **GND** to establish the idle state, and a diode to **VCC** as protection against ESD. They can be connected directly to the **LinkOut** pins of other transputers or TRAMs.

LinkSpeedA, LinkSpeedB These select the speeds of **Link0** and **Link1,2,3** respectively. Table 2.2 shows the possible combinations.

LinkSpeedA	LinkSpeedB	Link0	Link1,2,3
0	0	10 Mbits/s	10 Mbits/s
0	1	10 Mbits/s	20 Mbits/s
1	0	20 Mbits/s	10 Mbits/s
1	1	20 Mbits/s	20 Mbits/s

Table 2.2 Link speed selection

ClockIn A 5MHz input clock for the transputer. The transputer synthesises its own high frequency clocks. **ClockIn** should have a stability over time and temperature of 200ppm. **ClockIn** edges should be monotonic within the range 0.6V to 2.0V with a rise/fall time of less than 8ns.

¹ Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Dual-In-Line Transputer Modules (TRAMs)* and *Module Motherboard Architecture* which are included in Part 3 of this databook. The *Transputer Databook* may also be of use. This is available as a separate publication from INMOS.

Reset Resets the transputer, and other circuitry. **Reset** should be asserted for a minimum of 100ms. After **Reset** is deasserted a further 100ms should elapse before communication is attempted on any link. After this time, the transputer on this TRAM is ready to accept a boot packet on any of its links.

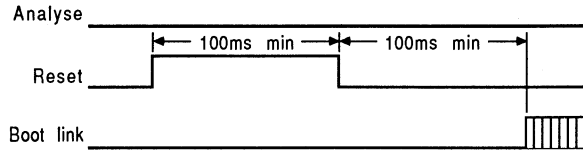


Figure 2.1 Reset timing

Analyse is used, in conjunction with **Reset**, to stop the transputer. It allows internal state to be examined so that the cause of an error may be determined. **Reset** and **Analyse** are used as shown in figure 2.2. A processor in analyse mode can be interrogated on any of its links.

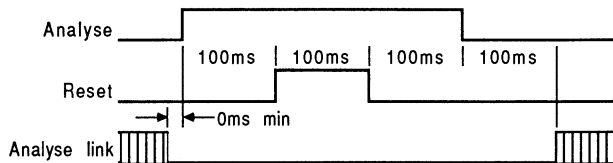


Figure 2.2 Analyse timing

notError An open collector output which is pulled low when the transputer asserts its Error pin. **notError** should be pulled high by a 10k Ω resistor to **VCC**. Up to 10 **notError** signals can be wired together. The combined error signal will be low when any of the contributing signals is low.

2.10.3 Ethernet Capabilities

The IMS B407 enables transputer based systems to connect to IEEE802.3 Local Area Networks (LANs). It enables transputer systems to communicate with other computers and with other transputer systems where a standard INMOS link connection would be unsuitable.

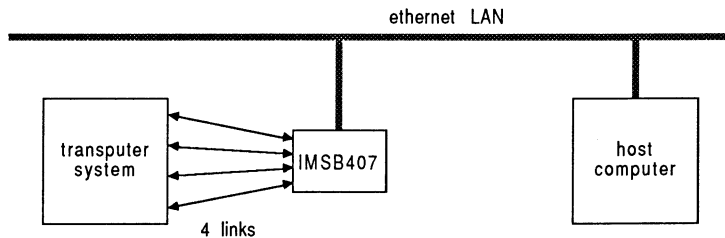


Figure 2.3 Typical Application

The IMS B407 can connect either to IEEE802.3 10BASE5 networks (Ethernet) or to IEEE802.3 10BASE2 networks (Cheapernet). The two types are network are logically identical, though the physical implementations are different.

Connecting to Ethernet (10BASE5)

In an ethernet (10BASE5) system, nodes are connected to the ethernet coax by means of an *media access unit* (MAU) and an *attachment unit interface* (AUI) cable. The MAU is a specially designed housing incorporating

a transceiver device. The MAU is clamped to the ethernet coax: it penetrates the coax making contact with the signal conductor without interrupting traffic on the LAN. The node is connected to the MAU transceiver by means of an AUI cable. Ethernet supports a maximum cable length (without repeaters) of 500 metres.

When connecting the IMS B407 to an ethernet system, the on board transceiver is not used. Instead, the AUI connector on the IMS B407 is used to connect to a separate MAU. Figure 2.4 shows how the IMS B407 should be connected to an ethernet system. It is intended that the 15-way D-type connector should be mounted in a suitable bulkhead or frontpanel on the equipment into which the IMS B407 is installed.

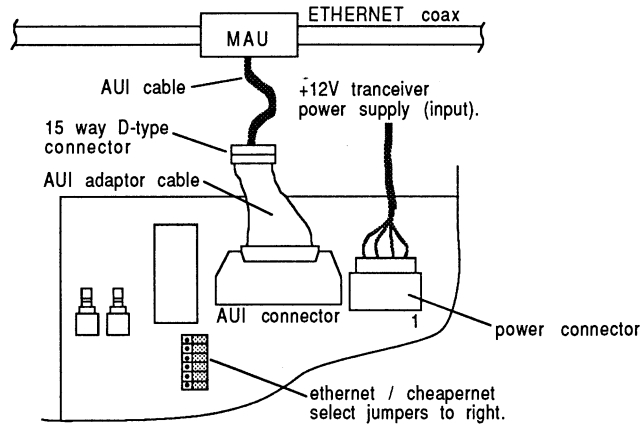


Figure 2.4 Connecting the IMS B407 to an Ethernet system

AUI connection

To reduce the overall height of the IMS B407, a 16-way male header is used instead of the standard 15-way D-type: the pinout of this connector is defined in table 2.3. A short length of adaptor cable is supplied for connection to a standard AUI cable. AUI cables should have a shielded, twisted pair for each signal or power pair: each pair should have a characteristic impedance of $78 \pm 5 \Omega$.

The RX_{\pm} , TX_{\pm} and $COLL_{\pm}$ signals on the AUI connector are transformer isolated. RX_{\pm} and $COLL_{\pm}$ are terminated with 80.4Ω .

Name	Function	Pin No.
COLL+	collision pair	11
COLL-		12
TX+	transmit pair	3
TX-		4
RX+	receive pair	7
RX-		8
+12V	power pair	13
0V		14
GND	shield	1,2,5,6,9,10,15,16

Table 2.3 AUI connector pinout

Jumper settings for Ethernet (10BASE5)

When using an offboard transceiver, all jumpers should be set to the right when viewing the board with the IMS T222 at the right hand end.

AUI Power

The ethernet specification requires that the AUI cable must supply +12V at 0.5A to the MAU. Since there is no +12V supply on the IMS B407, this must come from an external power source. A +12V power source capable of supplying 0.5A should be connected to the 4 way power connector on the IMS B407. The IMS B407 routes power from this connector to the AUI connector. The pinout of the power connector is shown in table 2.4.

Name	Function	Pin No.
+12V In	AUI power pair	1
0V In		2
GND	IMS B407 Power	3
+5V		4

Table 2.4 Power connector pinout

Pins 1 and 2 connect directly to the AUI connector power pair pins. Note that pins 3 and 4 are the IMS B407 power supply. These are nominally output pins which can be used to power a +5V to +12V DC-DC converter: the current drawn should not exceed 800mA. However, they can also be used to supply power to the IMS B407 *if the IMS B407 is not plugged into a motherboard.*

Connecting to Cheapernet (10BASE2)

In a cheapernet (10BASE2) system RG58A/U or RG58C/U coaxial cable is used; this being cheaper than the ethernet coax. Node connections are usually made with standard BNC T connectors. Thus, it is usually necessary to interrupt the LAN traffic and break the cable to connect a new node. The transceiver is also normally incorporated into the node so that a separate MAU and AUI cable are not required. Cheapernet supports a maximum cable length (without repeaters) of 185 metres.

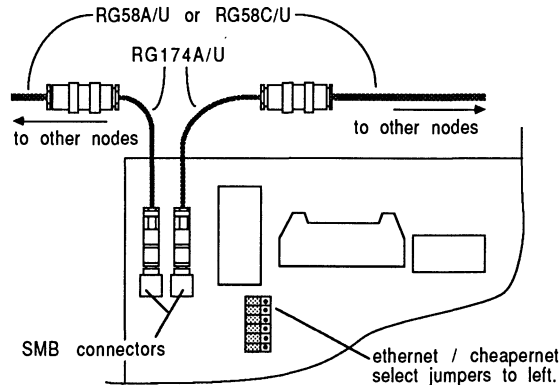


Figure 2.5 Connecting the IMS B407 to a Cheapernet system

When connecting the IMS B407 to a cheapernet system, the on board transceiver is used. The IMS B407 is connected to the LAN by means of two miniature coaxial connectors (SMB type). These miniature connectors are used to reduce the overall height of the IMS B407. Since they will not mate with the BNC connectors used in ethernet systems, two short lengths of adaptor cable are supplied. It is intended that the BNC ends of the adaptor cable are taken to insulating, through-bulkhead adaptors mounted on a suitable bulkhead or frontpanel of the equipment into which the IMS B407 is fitted. It is **very** important that the shields of the coaxial connectors are insulated from other conductors, including ground. Figure 2.5 shows how the IMS B407 should be connected to a cheapernet system.

The transceiver device used on the IMS B407 is the Am7996. It is fully isolated from the rest of the board by isolation transformers and is powered by an isolating DC-DC converter. Isolation is to 500V dc.

Jumper settings for Cheapernet (10BASE2)

When using the onboard transceiver, all jumpers should be set to the left when viewing the board with the IMS T222 at the right hand end.

2.10.4 Memory Map

The IMS T222 on the IMS B407 has access to 64 kbytes of memory. This is comprised of 4 kbytes of internal transputer memory and 60 kbytes of external SRAM. It also has memory mapped access to the two Am 7990 (LANCE) control registers. These occupy the top four byte locations in the memory map.

	Hardware byte address
IMS T222 on chip RAM	#8000 - #8FFF
External Static RAM	#9000 - #7FFB
Am7990 (LANCE) RDP	#7FFC
Am7990 (LANCE) RAP	#7FFE

Table 2.5 Memory map of the IMS B407

Table 2.5 shows the address map of the IMS B407 (the “#” sign indicates a hexadecimal number). Addresses range from #8000 through #0000 to #7FFF. The internal RAM on the IMS T222 occupies the first 4 Kbytes of address space. The Am 7990 (LANCE) occupies the top four bytes of memory. The internal RAM on the IMS T222 has a 50 nsec access cycle and the external SRAM has a 150 nsec access cycle. The LANCE has DMA to the external SRAM, DMA cycles are 600ns.

2.10.5 Using the IMS B407

The Ethernet interface on the IMS B407 is implemented with the Am 7990 (LANCE). The LANCE avoids loading the IMS T222 with frequent I/O operations by having direct memory access (DMA). The LANCE transmits Ethernet packets directly from a set of transmit buffers, and receives packets directly into a set of receive buffers. The programmer can place the transmit and receive buffers in any convenient areas of memory.

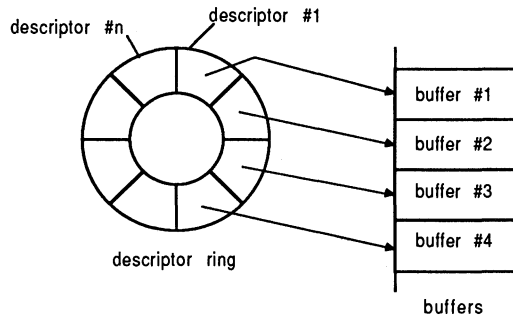


Figure 2.6 Data buffers and descriptors

Each buffer has an associated descriptor. The descriptors arbitrate buffer ownership between IMS T222 and LANCE and provide comprehensive error and status reporting for each packet received or transmitted. Each descriptor points to a single buffer. The descriptors are organised as two rings: a transmit ring and a receive ring. Descriptors in the transmit ring point to transmit buffers, descriptors in the receive ring point to receive buffers. Buffers and descriptors are used strictly in the order they appear in the ring. The descriptors can be placed in any convenient area of memory.

The LANCE is able to interrupt the IMS T222. Interrupts are generated for: reception of a packet, transmission of a packet, receive errors, and transmit errors.

In a typical application, software running on the IMS T222 would accept packets to be transmitted on one or more of its links and would output received packets on one or more of its links.

To transmit a packet, the IMS T222 simply has to place the packet in an empty transmit buffer and set up the descriptor for that buffer to indicate that it should be transmitted. The LANCE will transmit the buffer contents when it has transmitted all packets ahead of it in the descriptor ring. It then updates the descriptor contents to inform the IMS T222 of the packet status. It may also interrupt the IMS T222.

When the LANCE receives a packet from the Ethernet (or Cheapernet) it places the packet in the next available empty receive buffer. It then updates the descriptor for that receive buffer to indicate that it contains a received packet. It may also interrupt the IMS T222. When the IMS T222 has removed the received packet from the buffer it marks the descriptor as empty again.

Operation of the IMS B407 is the same regardless of whether it is connected to an Ethernet system or a Cheapernet system: the same software can be used with both types of network.

The LANCE has comprehensive test features including internal and external loopback tests, collision detection logic test, and CRC logic test.

2.10.6 Mechanical details

Figure 2.7 gives the vertical dimensions of an IMS B407 and Figure 2.8 is an outline drawing of the IMS B407. When the IMS B407 is mounted as shown in figure 2.7, the motherboard can be placed in a 0.8 inch pitch card cage without fouling adjacent boards.

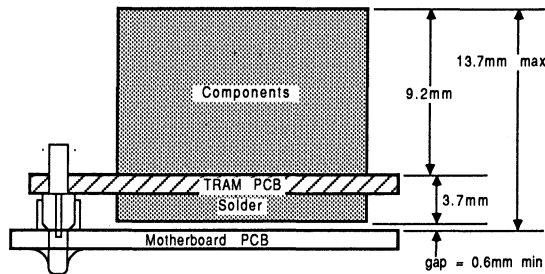


Figure 2.7 IMS B407 height specification

2.10.7 Installation

Since the IMS B407 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B407 may be supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

Plug the IMS B407 into the motherboard. Where the IMS B407 is being used with an INMOS motherboard, the yellow triangle marking pin 1 on the IMS B407 (see Figure 2.8) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot.

Should it be necessary to unplug the IMS B407, it is advised that it is gently levered out while keeping it as flat as possible. As soon as the IMS B407 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

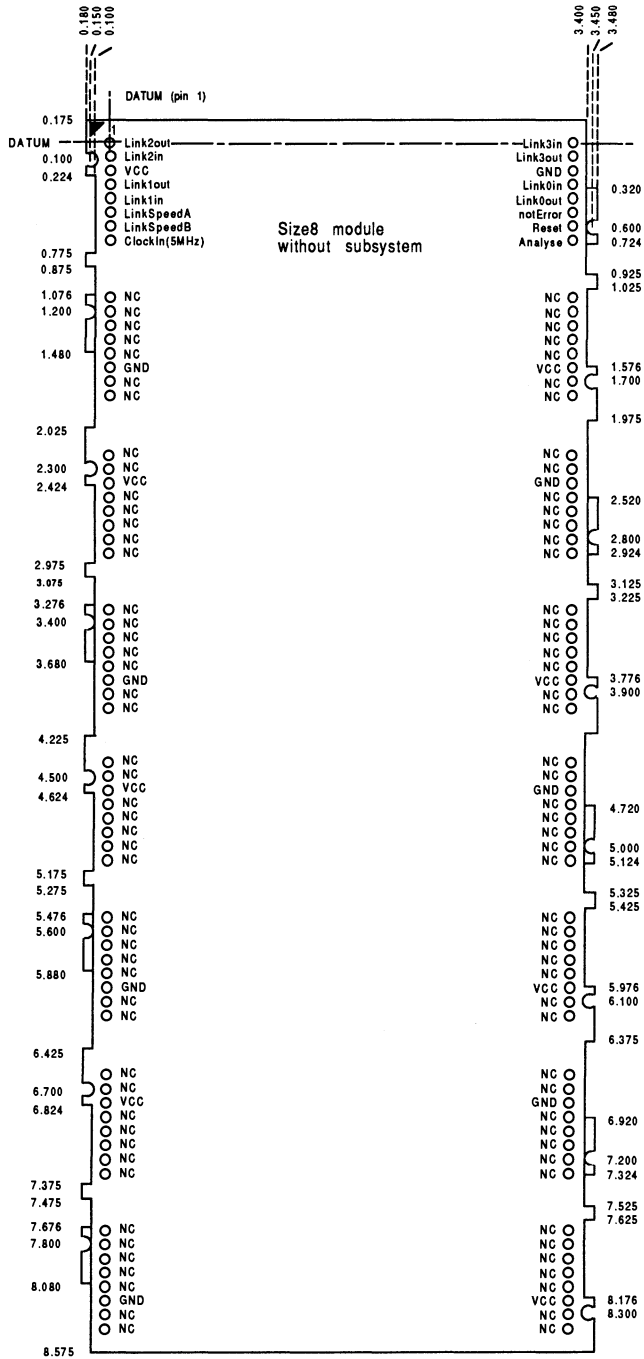


Figure 2.8 IMS B407 outline drawing (All dimensions in inches)

2.10.8 Specification

TRAM feature		Unit	Notes
Transputer type	IMS T222-20		
Number of transputers	1		
Number of INMOS serial links	4		
Amount of SRAM	64	kbyte	
Memory "wait states"	1		
Memory cycle time	150	ns	
Subsystem controller	No		
Peripheral circuitry	IEEE802.3 Interface		
Memory Parity	No		
Size (TRAM size)	8		
Length	3.66	inch	
Pitch between pins	3.30	inch	
Width	8.75	inch	
Component height above PCB	9.2	mm	1
Component height below PCB	3.7	mm	2
Weight	160	g	
Storage temperature	0–70	deg C	
Operating temperature	10–40	deg C	3
Power supply voltage (VCC)	4.75–5.25	Volt	
Power consumption	12	W	4

Table 2.6 IMS B407 specification

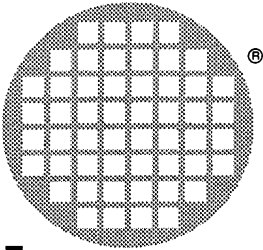
Notes:

- 1 This dimension is larger than is normally stated for TRAMs because of the requirement to connect to Ethernet.
- 2 This dimension includes the thickness of the PCB.
- 3 The figure quoted refers to the ambient air temperature.
- 4 The power consumption is the worst case value obtained when a sample of IMS B407 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.

2.10.9 Ordering Information

Description	Order Number
IMS B407 TRAM with IMS T222-20	IMS B407-1

Table 2.7 Ordering information



inmos

IMS B421

IEEE 488 GPIB TRAM

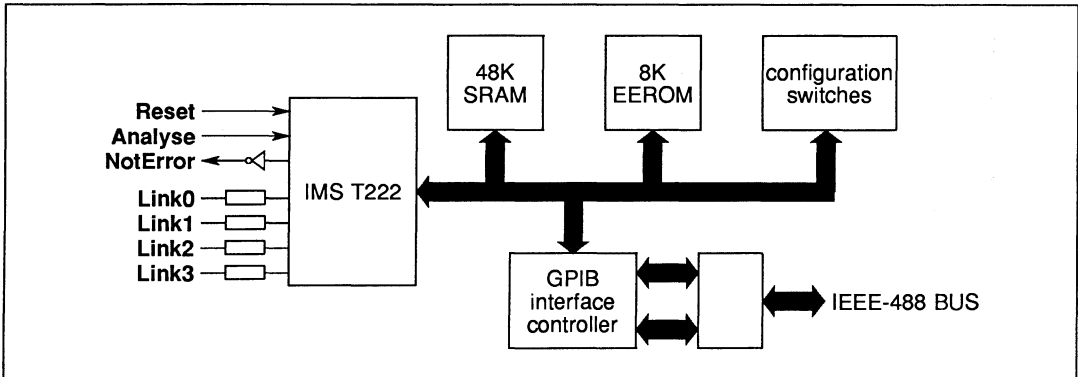
Size 4

FEATURES

- IMS T222 transputer
- 48 Kbytes of two-cycle RAM
- Full electrical compliance with IEEE-488 specification
- Size 4 TRAM
- Switchable GPIB bus address
- On-board non-volatile storage for configuration data
- Communicates via 4 INMOS links
- Designed to a published specification (*INMOS Technical Note 29*).

GENERAL DESCRIPTION

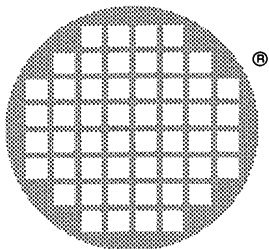
The GPIB TRAM allows IEEE-488 test and instrumentation systems to be directly connected to networks of transputers. The parallel interface permits high speed communication of control and measurement information, and the power of the transputer can provide sophisticated data analysis facilities. The user can define the characteristics of the GPIB interface in terms of address, etc., for maximum flexibility in system configuration.



Ordering Information

Description	Order Number
IMS B421 GPIB TRAM with T222-20	IMS B421-1

Table 1: Ordering information



inmos

IMS B422

SCSI TRAM

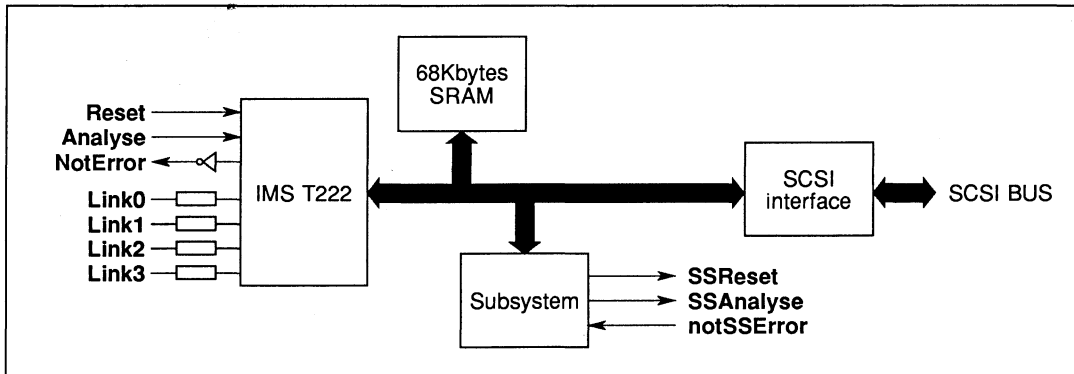
Size 2

FEATURES

- IMS T222-20 transputer
- 64 Kbytes of two-cycle memory
- SCSI bus interface (single ended drivers)
- Sustained SCSI transfer rates up to 1.5 MBytes/s
- Target or Initiator modes
- On-board, user removable SCSI bus terminators
- Subsystem port
- Size 2 TRAM
- Designed to a published specification (*INMOS Technical Note 29*).

GENERAL DESCRIPTION

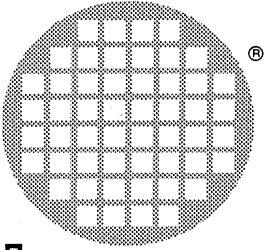
The SCSI TRAM acts as an interface between an INMOS link and the SCSI bus as defined in the ANSI X3.131-1986 standard. It allows transputer systems to connect to winchester disks, optical disks, and other peripherals via the SCSI bus. The SCSI TRAM consists of an IMS T222 16 bit transputer with 64 Kbytes of SRAM for program and data buffers. An intelligent interface device is used to implement the connection to the SCSI bus which allows common sequences to proceed without intervention from the IMS T222. Target and initiator modes are supported. On board removable SCSI bus terminators are provided. A standard subsystem port is implemented on the TRAM.



Ordering Information

Description	Order Number
IMS B422 SCSI TRAM with T222-20	IMS B422-1

Table 1: Ordering information



inmos

IMS B408

Frame store TRAM

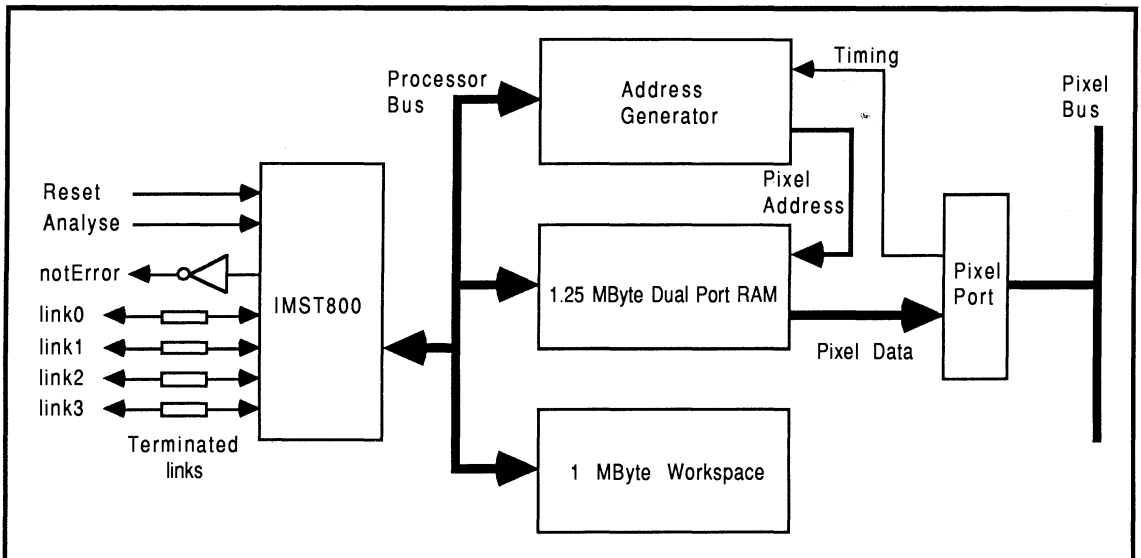
Size 8

FEATURES

- IMS T800 Transputer
- 1 Mbytes single wait-state work space DRAM
- 1.25 Mbytes single wait-state dual port DRAM
- Dual Port supports continuous data rates up to 100 Mbytes/s
- Communicates via 4 INMOS serial links (Selectable between 10 or 20 Mbits/s)
- Designed to a published specification (*INMOS Technical Note 29*)

GENERAL DESCRIPTION

The IMS B408 implements the drawing and image storage parts of a medium to high performance graphics system. It incorporates a powerful 32-bit microprocessor with on-chip FPU, 1 Mbyte of workspace RAM and 1.25 Mbyte of display RAM accessible to the processor and dual ported to the Pixel Port. The pixel port is capable of sustaining continuous data transmission at up to 100 Mbytes/sec, independently of the processor, and under control of an autonomous address generator. The IMS B408 supports both interlaced and non-interlaced displays of arbitrary resolution up to 1024×768 pixels. At lower resolutions multiple frame buffers are supported; e.g. 4 frame buffers of 640×480 pixels.



2.13 IMS B408 TRAM engineering data

2.13.1 Introduction

The IMS B408 is one of a range of INMOS TRANsputer Modules (TRAMs). In effect, TRAMs are board level transputers with a simple, standardised interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort. ¹

The IMS B408 is designed to be used with the IMS B409 display driver TRAM. When connected to an IMS B409 via the INMOS Pixel Bus (and a suitable video monitor) a complete drawing and display system is formed. System performance is increased simply by adding more IMS B408s.

The IMS B408 performs the drawing function in such a system. The graphics processor is an IMS T800; a fast 32 bit processor with on-chip FPU. Image data is drawn into 1.25 Mbyte of dual port RAM; a further 1 Mbyte of RAM is provided for program/data storage. Image data is output through the pixel bus pixel port under the control of the dual port address generator. The address generator is programmable and responds to system timing signals from the pixel bus.

2.13.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkspeedA,B	in	Transputer link speed selection	6,7

Table 2.1 IMS B408 Pin designations

Notes:

- Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- Details of the physical pin locations can be found in Fig. 2.4.

LinkOut0-3 Transputer link output signals. These outputs are intended to drive into transmission lines with a characteristic impedance of 100Ω. They can be connected directly to the **LinkIn** pins of other transputers or TRAMs.

LinkIn0-3 Transputer link input signals. These are the link inputs of the transputer. Each input has a 10KΩ resistor to **GND** to establish the idle state, and a diode to **VCC** as protection against ESD. They can be connected directly to the **LinkOut** pins of other transputers or TRAMs.

LinkSpeedA, LinkSpeedB These select the speeds of **Link0** and **Link1,2,3** respectively. Table 2.2 shows the possible combinations.

¹Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Dual-In-Line Transputer Modules (TRAMs)* and *Module Motherboard Architecture* which are included in Part 3 of this databook. The *Transputer Databook* may also be required. This is available as a separate publication from INMOS (72 TRN 203 01).

LinkSpeedA	LinkSpeedB	Link0	Link1,2,3
0	0	10 Mbits/s	10 Mbits/s
0	1	10 Mbits/s	20 Mbits/s
1	0	20 Mbits/s	10 Mbits/s
1	1	20 Mbits/s	20 Mbits/s

Table 2.2 Link speed selection

ClockIn A 5MHz input clock for the transputer. The transputer synthesises its own high frequency clocks. **ClockIn** should have a stability over time and temperature of 200ppm. **ClockIn** edges should be monotonic within the range 0.8V to 2.0V with a rise/fall time of less than 8ns.

Reset Resets the transputer, and other circuitry. **Reset** should be asserted for a minimum of 100ms. After **Reset** is deasserted a further 100ms should elapse before communication is attempted on any link. After this time, the transputer on this TRAM is ready to accept a boot packet on any of its links.

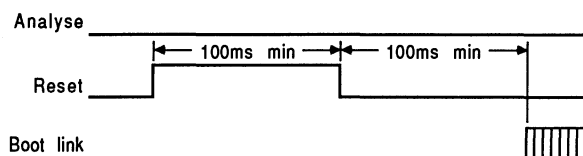


Figure 2.1 Reset timing

Analyse is used, in conjunction with **Reset**, to stop the transputer. It allows internal state to be examined so that the cause of an error may be determined. **Reset** and **Analyse** are used as shown in figure 2.2. A processor in analyse mode can be interrogated on any of its links.

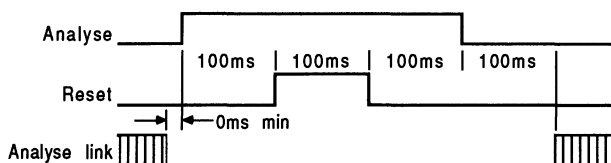


Figure 2.2 Analyse timing

notError An open collector output which is pulled low when the transputer asserts its Error pin. **notError** should be pulled high by a 10K Ω resistor to VCC. Up to 10 **notError** signals can be wired together. The combined error signal will be low when any of the contributing signals is low.

2.13.3 Pixel Port signals

The IMS B408 has a pixel data port in addition to the usual TRAM signals. This enables the TRAM to connect via the INMOS pixel bus to an IMS B409 display TRAM; and possibly several other IMS B408s. The pinout is defined in Table 2.3. The pixel bus uses 60-way IDC connectors and flat ribbon cable.

Pin No.	In/Out	Pin	Pin	In/Out	Pin No.
1		GND	D0	out	2
3	out	D1	GND		4
5	out	D2	D3	out	6
7		GND	D4	out	8
9	out	D5	GND		10
11	out	D6	D7	out	12
13		GND	D8	out	14
15	out	D9	GND		16
17	out	D10	D11	out	18
19		GND	D12	out	20
21	out	D13	GND		22
23	out	D14	D15	out	24
25		GND	D16	out	26
27	out	D17	GND		28
29	out	D18	D19	out	30
31		GND	D20	out	32
33	out	D21	GND		34
35	out	D22	D23	out	36
37		GND	D24	out	38
39	out	D25	GND		40
41	out	D26	D27	out	42
43		GND	D28	out	44
45	out	D29	GND		46
47	out	D30	D31	out	48
49		GND	notSEQclk	in	52
51		GND	notRAMclk	in	50
53		GND	notFieldSync	in	54
55		GND	notEarlyBlank	in	56
57		GND	notEvenField	in	58
59		GND	SysReady	in/out	60

Table 2.3 Pixel Bus pin designations

D0 - 31 Pixel data is output on a 32 bit bus. Transitions occur on the falling edge of **notSEQclk**. The data bus is open collector and carries inverted data. This allows data from different serial port modules to be ORed on the Pixel Bus.

notSEQclk A continuous input clock used as the timing reference by the dual port address generator. It has a maximum frequency of 25MHz. Data and control strobes transitions are synchronised to the falling edge of **notSEQclk**.

notRAMclk Used to clock pixel data from the dual port RAM onto the pixel bus. It is the same frequency and phase as **notSEQclk** but is gated so that it does not run during blanking. Thus, no data is lost during blanking. In the off state it is high.

notEarlyBlank A time-advanced version of the display blanking signal. Used by the dual port address generator as an early warning of when pixel data will be required and of when it should be turned off.

notFieldSync Resets the dual port address generator at the start of each field. Low during field flyback (vertical blanking).

notEvenField Used by the dual port address generator to ensure that the pixel data for the correct display field is output when generating an interlaced display. A low on this signal indicates the even field of the odd/even field pair making up an interlaced frame.

SysReady This acts as a synchronisation mechanism for multiple modules. The IMS B408 has a writeable READY bit which drives an open collector output onto this wire; it also monitors its state. Only when all IMS B408s in a system have written 1 (ready) to their READY bits will SYSREADY be 1. This can be used to EVENT (interrupt) the IMS T800.

Electrical Specification

The open collector drivers used for the data bus are capable of sinking 64mA and must be pulled up by an external resistor network. This network is part of the pixel data input structure on the IMS B409.

The clock and control inputs have 4K7 Ω pull up resistors to establish an idle condition on each input when the bus is disconnected.

2.13.4 Memory Map

There are 2304 Kbytes of memory. This is comprised of 4 Kbytes of internal transputer memory and 2300 Kbytes of external DRAM. The upper 1280 Kbytes is dual ported to the pixel port. The lower 1024 Kbytes would normally be used for program storage and the dual ported area as a drawing area (frame buffer). Table 2.4 shows how the memory is mapped into the address space of the IMS T800 (the “#” sign indicates a hexadecimal number).

	Byte address	Cycle Time
IMS T800 on chip RAM	#80000000 - #80000FFF	50ns
External Workspace RAM	#80001000 - #800FFFFF	200ns
Dual port RAM	#80100000 - #8023FFFF	200ns

Table 2.4 Memory map of the IMS B408

2.13.5 Pixel Port control registers

There are a small number of control registers associated with the pixel port and its address generator. These registers are located as shown in Table 2.5.

Register	Byte address
Display Start (write only)	#00000000
Ready (read,write)	#00040000
SysReady (read only)	#00080000
Interlace Enable (read,write)	#000C0000
Event Mode (read,write)	#00100000
Output Enable (read,write)	#00140000

Table 2.5 Control register locations

Display Start This registers holds the address of the pixel at the top left hand corner of the displayed image. It can be used to implement flipping between multiple drawing buffers. Buffers must start on 64 kbyte boundaries.

Interlace Enable Selects an interlaced or non-interlaced display. Writing 1 causes the address generator to produce interlace addressing; writing 0 causes it to produce non-interlaced addressing.

Event Mode Selects the EVENT (interrupt) source to be either FieldSync or SysReady.

Output Enable Enables and disables the pixel port output buffers. Writing 1 enables the data output buffers; writing 0 disables them.

Ready Writing 0 drives SysReady low; writing 1 allows it to be pulled high.

SysReady is a read only location which reflects the condition of the SysReady wire. Bit 0 is read as 0 if SysReady is low; 1 if SysReady is high.

2.13.6 Mechanical details

Figure 2.3 indicates the vertical dimensions of a single IMS B408 and Figure 2.4 shows the outline drawing of the IMS B408. Note that the component height includes the height taken up by a cable plugged into the pixel port connector. This means that the IMS B408 on a motherboard occupies more than one card slot in a 0.8in pitch card cage.

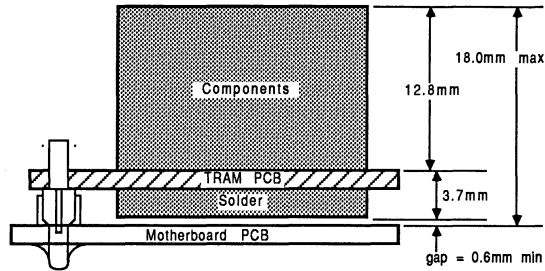


Figure 2.3 IMS B408 height specification

2.13.7 Installation

Since the IMS B408 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B408 may be supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

Plug the IMS B408 into the motherboard. Where the IMS B408 is being used with an INMOS motherboard, the copper triangle marking pin 1 on the IMS B408 (see Figure 2.4) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot.

Should it be necessary to unplug the IMS B408, it is advised that it is gently levered out while keeping it as flat as possible. As soon as the IMS B408 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

2.13.8 Specification

TRAM feature		Unit	Notes
Transputer type	IMS T800-20		
Number of transputers	1		
Number of INMOS serial links	4		
Amount of DRAM	2.25	Mbyte	
DRAM "wait states"	1		
Memory cycle time	200	ns	
Subsystem controller	No		
Peripheral circuitry	Pixel Port		
Parity	No		
Size (TRAM size)	8		
Length	3.66	inch	
Pitch between pins	3.30	inch	
Width	8.75	inch	
Component height above PCB	12.8	mm	1
Component height below PCB	3.0	mm	2
Weight	215	g	
Storage temperature	0-70	deg C	
Operating temperature	10-40	deg C	3
Power supply voltage (VCC)	4.75-5.25	Volt	
Power consumption	18	W	4

Table 2.6 IMS B408 specification

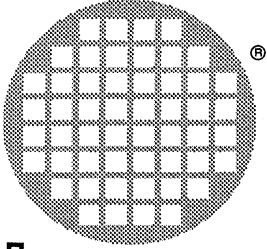
Notes:

- 1 This dimension is larger than is normally stated for TRAMs because of the requirement to connect to the pixel bus.
- 2 This dimension includes the thickness of the PCB.
- 3 The figure quoted refers to the ambient air temperature.
- 4 The power consumption is the worst case value obtained when a sample of IMS B408 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.

2.13.9 Ordering Information

Description	Order Number
IMS B408 TRAM with IMS T800-20	IMS B408-1

Table 2.7 Ordering information



inmos

IMS B409

Display TRAM

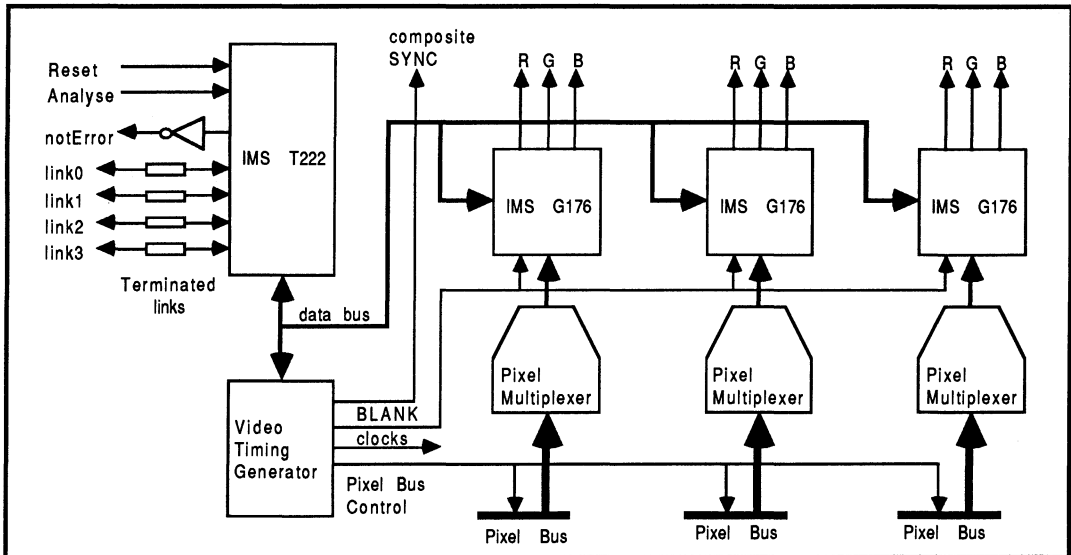
Size 8

FEATURES

- IMS T222, 16-bit Transputer
- Video timing generator
- Pixel rates up to 64 MHz
- 8 or 18 bit pixels
- 3 IMS G176 colour look-up tables
- Designed to a published specification (*INMOS Technical Note 29*)

GENERAL DESCRIPTION

The IMS B409 implements the timing generation and display driver parts of a medium to high performance graphics system. It consists of three pixel channels and a programmable video timing generator (VTG), controlled by an IMS T222. Each pixel channel consists of a 4-1 byte multiplexer and an IMS G176 colour look-up table (CLUT). Input to each pixel channel is by a separate pixel bus input and each channel generates a set of RGB outputs. The IMS B409 supports both interlaced and non-interlaced displays of arbitrary resolution up to a maximum dot rate of 64MHz.



2.14 IMS B409 TRAM engineering data

2.14.1 Introduction

The IMS B409 is one of a range of INMOS TRANsputer Modules (TRAMs). In effect, these TRAMs are board level transputers with a simple, standardised interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort. ¹

The IMS B409 is designed to be used in conjunction with one or more IMS B408 frame store TRAMs. When connected to an IMS B408 via the INMOS Pixel Bus (and a suitable video monitor) a complete drawing and display system is formed. System performance is increased simply by adding more IMS B408s.

The IMS B409 has three pixel channels. Each channel inputs a 32 bit wide pixel stream from an INMOS Pixel Bus and processes it into a form suitable for display by a colour monitor. The IMS B409 also generates system timing and control signals and outputs them on each pixel bus.

2.14.2 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5 MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkSpeedA,B	in	Transputer link speed selection	6,7

Table 2.1 IMS B409 Pin designations

Notes:

- Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- Details of the physical pin locations can be found in Fig 2.4.

LinkOut0-3 Transputer link output signals. These outputs are intended to drive into transmission lines with a characteristic impedance of 100Ω. They can be connected directly to the **LinkIn** pins of other transputers or TRAMs.

LinkIn0-3 Transputer link input signals. These are the link inputs to the transputer. Each input has a 10kΩ resistor to **GND** to establish the idle state, and a diode to **VCC** as protection against ESD. They can be connected directly to the **LinkOut** pins of other transputers or TRAMs.

LinkSpeedA, LinkSpeedB These select the speeds of **Link0** and **Link1,2,3** respectively. Table 2.2 shows the possible combinations.

ClockIn A 5 MHz input clock for the transputer. The transputer synthesises its own high frequency clocks. **ClockIn** should have a stability over time and temperature of 200ppm. **ClockIn** edges should be monotonous within the range 0.8V to 2.0V with a rise/fall time of less than 8ns.

¹ Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Dual-In-Line Transputer Modules (TRAMs)* and *Module Motherboard Architecture* which are included in Part 3 of this databook. The *Transputer Databook* may also be required. This is available as a separate publication from INMOS (72 TRN 203 01).

LinkSpeedA	LinkSpeedB	Link0	Link1,2,3
0	0	10 Mbits/s	10 Mbits/s
0	1	10 Mbits/s	20 Mbits/s
1	0	20 Mbits/s	10 Mbits/s
1	1	20 Mbits/s	20 Mbits/s

Table 2.2 Link speed selection

Reset Resets the transputer, and other circuitry. **Reset** should be asserted for a minimum of 100ms. After **Reset** is deasserted a further 100ms should elapse before communication is attempted on any link. After this time, the transputer on the IMS B409 is ready to accept a boot packet on any of its links.

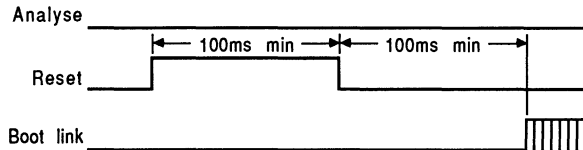


Figure 2.1 Reset timing

Analyse is used, in conjunction with **Reset**, to stop the transputer. It allows internal state to be examined so that the cause of an error may be determined. **Reset** and **Analyse** are used as shown in figure 2.2. A processor in analyse mode can be interrogated on any of its links.

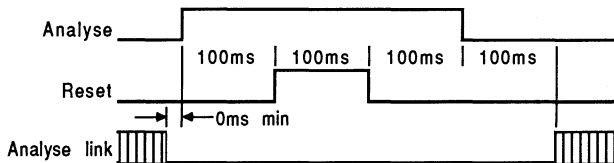


Figure 2.2 Analyse timing

notError An open collector output which is pulled low when the transputer asserts its Error pin. **notError** should be pulled high by a 10k Ω resistor to **VCC**. Up to 10 **notError** signals can be wired together. The combined error signal will be low when any of the contributing signals is low.

2.14.3 Pixel Bus connectors

The IMS B409 has three pixel data ports in addition to the usual TRAM signals. This enables the IMS B409 to connect via the INMOS pixel bus to one or more IMS B408s. The Pixel Bus uses 60-way IDC connectors and flat ribbon cable; the pinout of each port is defined in Table 2.3. The clock and control outputs are driven by high current buffers capable of driving into 100 Ω loads.

D0 - 31 Pixel input data is latched on the falling edge of **notSEQclk**. The data bus is open collector and carries inverted data. This allows data from different serial port modules to be ORed on the Pixel Bus. Each data input is terminated with 330 Ω to **VCC** and 470 Ω to **GND**.

notSEQclk A continuous output clock for use as the system timing reference; it has a maximum frequency of 25 MHz. Data and control strobes transitions are synchronised to the falling edge of **notSEQclk**.

notRAMclk An output clock of the same frequency and phase as **notSEQclk** but gated so that it does not run during display blanking. It is used to clock pixel data from the IMS B408s onto the pixel bus so that no data is lost during blanking. In the off state it is high.

Pin No.	In/Out	Pin	Pin	In/Out	Pin No.
1		GND	D0	in	2
3	in	D1	GND		4
5	in	D2	D3	in	6
7		GND	D4	in	8
9	in	D5	GND		10
11	in	D6	D7	in	12
13		GND	D8	in	14
15	in	D9	GND		16
17	in	D10	D11	in	18
19		GND	D12	in	20
21	in	D13	GND		22
23	in	D14	D15	in	24
25		GND	D16	in	26
27	in	D17	GND		28
29	in	D18	D19	in	30
31		GND	D20	in	32
33	in	D21	GND		34
35	in	D22	D23	in	36
37		GND	D24	in	38
39	in	D25	GND		40
41	in	D26	D27	in	42
43		GND	D28	in	44
45	in	D29	GND		46
47	in	D30	D31	in	48
49		GND	notSEQclk	out	52
51		GND	notRAMclk	out	50
53		GND	notFieldSync	out	54
55		GND	notEarlyBlank	out	56
57		GND	notEvenField	out	58
59		GND	SysReady		60

Table 2.3 Pixel Bus pin designations

notEarlyBlank A time-advanced version of the display blanking signal. It provides early warning of when pixel data will be required and of when it should be turned off.

notFieldSync Output low during field flyback (vertical blanking) to indicate the start of a new field.

notEvenField For use in systems producing interlaced displays; e.g. TV standard displays. A low on this signal indicates that pixel data for the even field of the odd/even field pair making up an interlaced frame should be placed on the pixel bus. Changes state on the falling edge of **notFieldSync**.

SysReady Used as a synchronisation mechanism by multiple IMS B408 frame store modules. It is neither driven nor monitored by the IMS B409 but is common between the three pixel channel bus connectors to support the synchronisation mechanism.

2.14.4 The Pixel channels

The IMS B409 has three pixel channels: A, B and C. Each channel consists of a 4-1 byte multiplexer and an IMS G176 colour look-up table (CLUT). Input to a channel is through a pixel bus connector, output is from a set of RGB video outputs. There are two operating modes.

8 bits/pixel mode

Each channel accepts 32 bit pixel data from a pixel bus connector at 1/4 the pixel rate. This is multiplexed down to an 8 bit wide stream at pixel rate which is fed to the CLUT pixel data inputs. Thus, the pixel bus only runs at 1/4 the pixel rate which may be up to 64 MHz. Each channel can provide a separate display with up to 256 colours on each screen. It is not necessary to use all three channels. Since the three channels are synchronised it is possible to use each channel to generate one of the RGB primaries. Skew between any two pixel channels on the same IMS B409 is less than 5ns. Each channel would be connected by a separate pixel bus to one or more IMS B408s.

18 bits/pixel mode

In this mode the IMS B409 provides a single display of up to 262144 colours. This mode allows a full colour display to be produced by an IMS B409 with a single IMS B408. The pixel bus runs at the pixel rate which is therefore limited to 25 MHz. Each pixel requires a 32 bit word to be supplied to the channel A pixel bus input. The least significant byte is routed direct to the channel A CLUT, the second least significant byte is routed direct to the channel B CLUT, and the third least significant byte is routed direct to the channel C CLUT. Each CLUT is used to generate one of the RGB colour primaries. Thus, a single input word specifies directly the red, green and blue components of a pixel. Skew between any two pixel channels on the same IMS B409 is less than 5ns.

The colour look-up tables

Each of these devices combines a 256 word, 18 bit wide RAM and three 6 bit DACs. 8 bit data applied to the device's pixel inputs addresses a location in the RAM. 6 bits of the addressed data are applied to each of the DACs which generate the red, green, and blue (RGB) outputs. Thus, the device can display up to 256 colours, selectable from a palette of 262144. The RAM contents are writeable and readable by the IMS T222.

Video Outputs

Each pixel channel has a set of RGB outputs brought out on three SMB connectors. The outputs are current sources with 75Ω termination and will drive 1V pk-pk into a 75Ω load. The outputs are d.c. coupled: 0.3V is blanking level and 1.0V (on load) is peak white.

Sync is not composited with the video signals but is available from a separate sync output (also an SMB connector). The sync output will also drive into 75Ω and is d.c. coupled: 5V is the idle level, sync pulses are 0V.

2.14.5 Memory Map

The IMS B409 is able to access 4 kbytes of internal transputer memory. This is sufficient memory to contain the small amount of code and data required to set up the colour look-up tables and the VTG. The internal memory on the IMS T222 has a 50ns access cycle time; i.e. a single processor cycle. The IMS T222 has a 64 kbyte address space with addresses ranging from #8000 to #7FFF where # indicates a hexadecimal number.

	Byte Address
IMS T222 internal RAM	#8000-#8FFF

Table 2.4 IMS B409 memory location

Pixel Channel Mode select

The pixel channel mode is set by writing to the Pixel Channel Mode Select register. Writing 1 selects multiplexed (8 bits/pixel) mode: writing 0 selects non-multiplexed (18 bits/pixel) mode. The register location is given in table 2.5. This register is write only.

Register	Byte Address
Channel Mode Select	#B000

Table 2.5 Channel Mode Select register location

The video timing generator

The video timing generator is an NEC D7220. It is mapped into the IMS T222's address space as shown in Table 2.6. It is used only as a programmable timing generator and performs no drawing functions. Line frequency, field frequency and resolution can be programmed (horizontal resolution must be a multiple of 64 pixels) and displays may be either interlaced or non-interlaced.

Register	Byte address
Parameter FIFO (write only)	#A000
Status Register (read only)	#A000
Command FIFO (write only)	#A002
FIFO read (read only)	#A002

Table 2.6 NEC D7220 register locations

The Colour look-up tables

Ordinary accesses to the CLUT registers should be made at the addresses shown in table 2.7. These registers are mapped as the lower 8 bits of a 16 bit word addressed at that location. They can be written and read either as 16 bit words or as bytes addressed at the given locations. If written as 16 bit words, the upper 8 bits are ignored; if read as 16 bit words, the upper 8 bits are read undefined.

Register	Byte Address
Channel A Pixel Address (write mode)	#0000
Channel A Colour Value	#0400
Channel A Pixel Mask	#0800
Channel A Pixel Address (read mode)	#0C00
Channel B Pixel Address (write mode)	#1000
Channel B Colour Value	#1400
Channel B Pixel Mask	#1800
Channel B Pixel Address (read mode)	#1C00
Channel C Pixel Address (write mode)	#2000
Channel C Colour Value	#2400
Channel C Pixel Mask	#2800
Channel C Pixel Address (read mode)	#2C00

Table 2.7 IMS G176 registers

Each CLUT has a single Pixel Address register which is addressable at two locations. Writing a pixel address to the first, places the CLUT in colour value write mode. Writing a pixel address to the second places the CLUT in colour value read mode. Reading either location returns the same value.

Block moves to and from the colour value registers should be made to the regions defined in table 2.8. In this region, the colour value register appears as an 8 bit wide register at each byte address. Thus, byte arrays of colour values can be block copied to and from these areas. Correct results for block writes are not guaranteed for pixel clock speeds of less than 16 MHz. Correct results for block reads are not guaranteed for pixel clock speeds of less than 28 MHz.

Register	Byte Address
Channel A Colour Value	#4400-#47FF
Channel B Colour Value	#5400-#57FF
Channel C Colour Value	#6400-#67FF

Table 2.8 IMS G176 block move areas

2.14.6 Mechanical details

Figure 2.3 indicates the vertical dimensions of a single IMS B409 and Figure 2.4 shows the outline drawing of the IMS B409. Note that the component height includes the height taken up by a cable plugged into a pixel bus input. This means that the IMS B409 on a motherboard occupies more than one card slot in a 0.8in. pitch card cage.

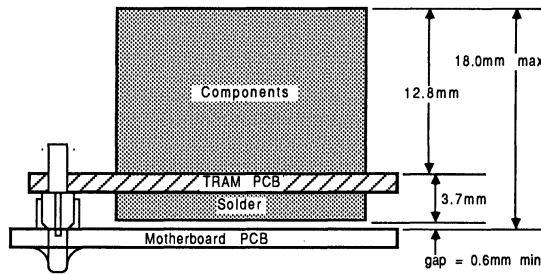


Figure 2.3 IMS B409 height specification

2.14.7 Installation

Since the IMS B409 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B409 may be supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

Plug the IMS B409 into the motherboard. Where the IMS B409 is being used with an INMOS motherboard, the yellow triangle marking pin 1 on the IMS B409 (see Figure 2.4) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot.

Should it be necessary to unplug the IMS B409, it is advised that it is gently levered out while keeping it as flat as possible. As soon as the IMS B409 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

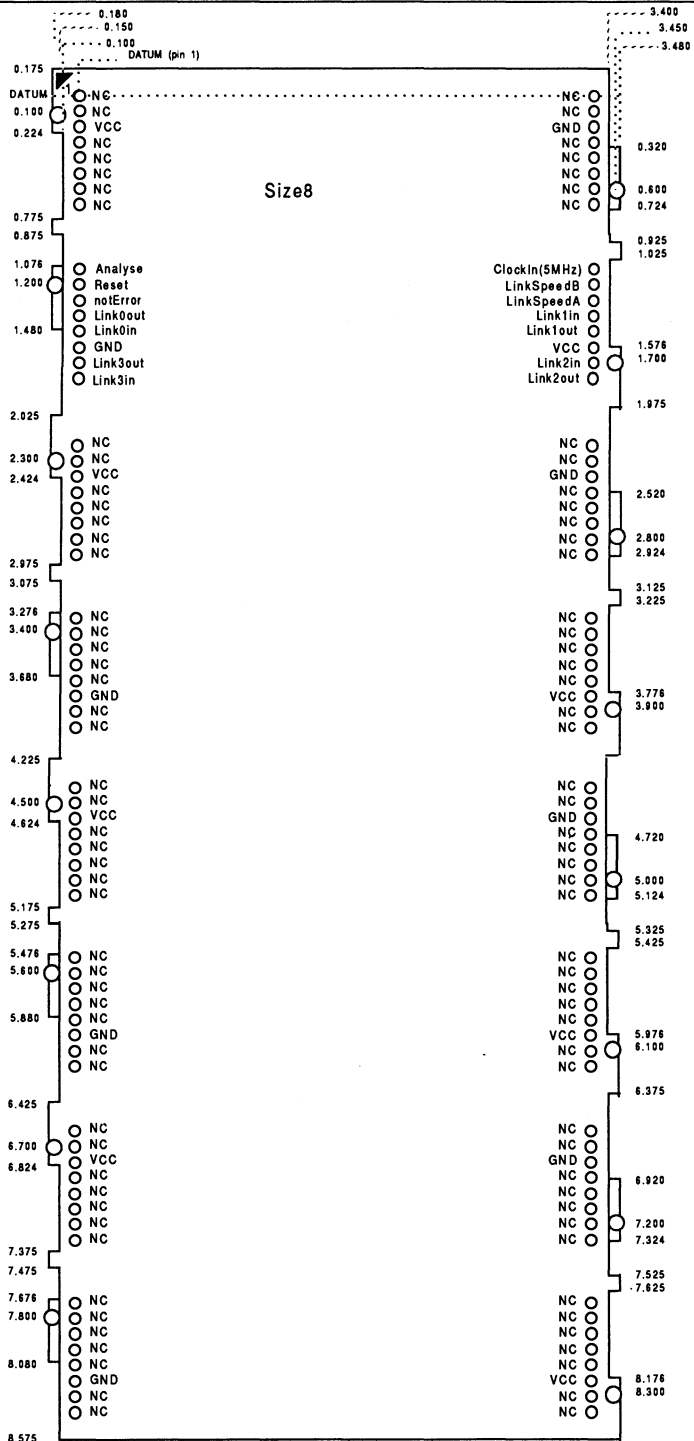


Figure 2.4 IMS B409 outline drawing (All dimensions in inches)

2.14.8 Specification

TRAM feature		Unit	Notes
Transputer type	IMS T222-20		
Number of transputers	1		
Number of INMOS serial links	4		
RAM size	4	kbyte	
Memory cycle time	50	ns	
Subsystem controller	No		
Peripheral circuitry	VTG		
Parity	No		
Size (TRAM size)	8		
Length	3.66	inch	
Pitch between pins	3.30	inch	
Width	8.75	inch	
Component height above PCB	12.8	mm	1
Component height below PCB	3.0	mm	2
Weight	185	g	
Storage temperature	0-70	deg C	
Operating temperature	10-40	deg C	3
Power supply voltage (VCC)	4.75-5.25	Volt	
Power consumption	18	W	4

Table 2.9 IMS B409 specification

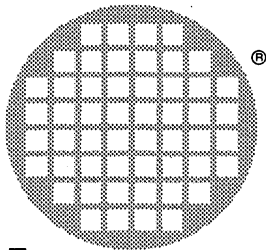
Notes:

- 1 Since the IMS B409 makes use of IDC connectors for the pixel bus, this dimension is larger than is normally stated for TRAMs.
- 2 This dimension includes the thickness of the PCB.
- 3 The figure quoted refers to the ambient air temperature.
- 4 The power consumption is the worst case value obtained when a sample of IMS B409 TRAMs were tested (running a program that utilised all four links and accessed memory simultaneously) at a supply voltage (VCC) of 5.25 V.

2.14.9 Ordering Information

Description	Order Number
IMS B409 TRAM with IMS T222-20	IMS B409-1

Table 2.10 Ordering information



inmos

IMS B419

Integrated graphics TRAM

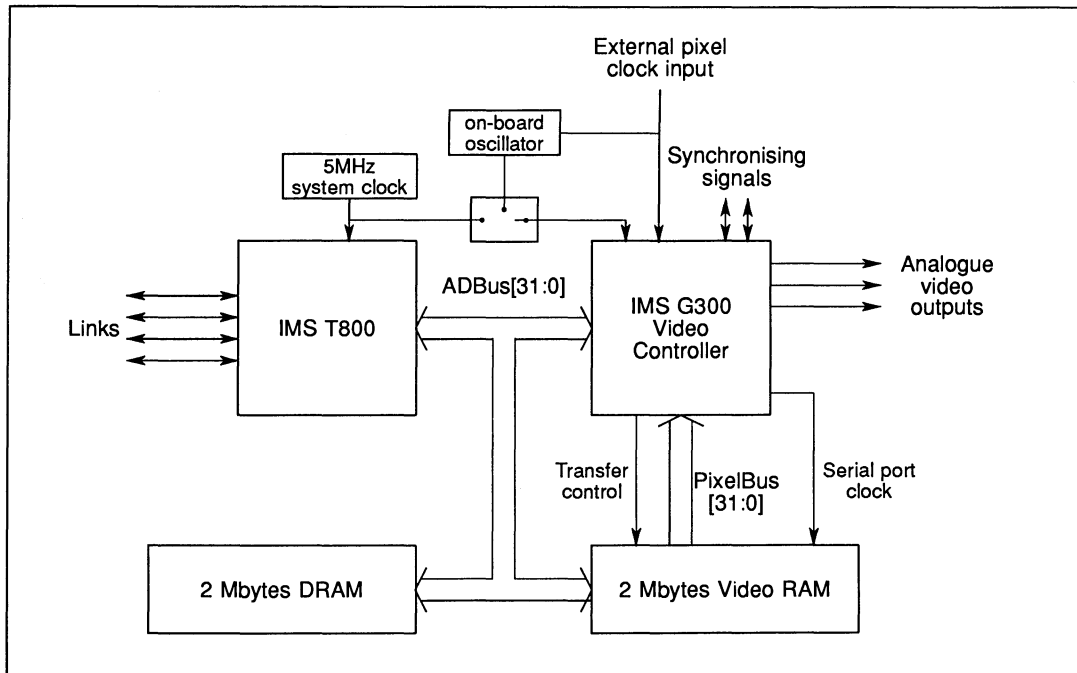
Size 6

FEATURES

- IMS T800 32 bit Transputer
- IMS G300 Colour Video Controller
- 2 Mbytes of four cycle DRAM
- 2 Mbytes of four cycle VRAM
- Huge variety of software selectable screen formats
- Pixel rates 70 to 110 MHz @ 8 bit/pixel
- Communicates via 4 INMOS serial links (Selectable between 10 or 20 Mbits)
- Size 6 TRAM
- Designed to a published specification (*INMOS technical Note 29*)

GENERAL DESCRIPTION

The IMS B419 incorporates the IMS G300 Colour Video Controller with the IMS T800 32 bit Floating Point Transputer to form a high performance graphics system. Two Mbytes of four cycle DRAM provides a general purpose store sufficient to run applications such as the X-window system, it also allows Transputer Development System (TDS) to be resident on board. Two Mbytes of Video RAM provide arbitrary screen resolutions up to a maximum of 1280 x 1024 8 bit/pixel with unrestricted screen formats at resolutions below this.



2.15 IMS B419 TRAM engineering data

2.15.1 Introduction

The IMS B419 is one of a range of INMOS TRAnsputer Modules (TRAMs). In effect, TRAMs are board level transputers with a simple, standardised interface. They integrate processor, memory and peripheral functions allowing powerful, flexible, transputer based systems to be produced with the minimum of design effort. ¹

The IMS B419 G300 CVC Graphics TRAM implements a complete high performance graphics subsystem. The frame store consists of 2 Mbytes of dual ported Video RAM which supports displays of arbitrary resolution at 8 bit/pixel. The resolution of the system is only limited by the CVCs maximum dot rate and the access time of the serial port on the VRAM. The IMS B419 supports a dot rate up to 110 MHz, the speed of the CVC. The CVC is configured by the on board IMS T800 which is provided with 2 Mbytes of 200ns cycle DRAM. This store is available for screen manipulation workspace and general program memory. The processor can be used to implement graphic primitives directly or as an intelligent channel receiving data from an array via its four bidirectional links at data rates of up to 10 Mbytes/sec. This makes the IMS B419 useful for applications as diverse as an add-on accelerator for a PC or a Macintosh, as part of an embedded system in industrial control, or as a graphics output for a 3D graphical supercomputer.

2.15.2 Screen sizes

Screen sizes can be set by writing to a few registers in the G300 CVC, and be chosen to suit the application. Suppose, for instance, an 8.5 x 11 sheet of paper (in landscape), represented by a screen with 100 pixels per inch. This would need an 1100 x 850 display, a format not normally available from a hardware solution. The G300 gives a line width in multiples of 4 pixels, which makes it simple to produce this screen. As well as producing special screens such as 11 x 8.5, many of the standard screens can also be produced; indeed the user can switch between screen formats, the display clock frequency, and even the source of the input clock, all by simply changing the G300 registers and other registers on the board by software.

Some examples of screen sizes that are possible are given in Table 2.1. All the screens in the table are for 8 bits per pixel.

Screen Size	Pixels	Aspect Ratio	Interlace
CGA	320 x 240	1.333	no
EGA	640 x 350	1.829	no
VGA	640 x 480	1.333	no
Enh VGA	800 x 600	1.333	no
Ext VGA	1024 x 768	1.333	no
11 x 8.5	1100 x 850	1.294	no
11 x 8.5	1164 x 900	1.293	no
	1024 x 1024	1.0	no
	1280 x 1024	1.25	no
A5	1216 x 860	1.414	no
PAL	768 x 575	1.333	yes
NTSC	668 x 501	1.333	yes

Table 2.1 A selection of possible screen sizes

¹Further details of the TRAM/motherboard philosophy and the full electrical and mechanical specification of TRAMs can be found in technical notes *Dual-In-Line Transputer Modules (TRAMs)* and *Module Motherboard Architecture* which are included in Part 3 of this databook. The *Transputer Databook* may also be required. This is available as a separate publication from INMOS (72 TRN 203 01).

2.15.3 Pin descriptions

Pin	In/Out	Function	Pin No.
System Services			
VCC, GND		Power supply and return	3,14
ClockIn	in	5 MHz clock signal	8
Reset	in	Transputer reset	10
Analyse	in	Transputer error analysis	9
notError	out	Transputer error indicator (inverted)	11
Links			
LinkIn0-3	in	INMOS serial link inputs to transputer	13,5,2,16
LinkOut0-3	out	INMOS serial link outputs from transputer	12,4,1,15
LinkSpeedA,B	in	Transputer link speed selection	6,7

Table 2.2 IMS B419 Pin designations

Notes:

- 1 Signal names are prefixed by **not** if they are active low; otherwise they are active high.
- 2 Details of the physical pin locations can be found in Fig. 2.5.

LinkOut0-3 Transputer link output signals. These outputs are intended to drive into transmission lines with a characteristic impedance of 100Ω. They can be connected directly to the **LinkIn** pins of other transputers or TRAMs.

LinkIn0-3 Transputer link input signals. These are the link inputs of the transputer. Each input has a 10kΩ resistor to **GND** to establish the idle state, and a diode to **VCC** as protection against ESD. They can be connected directly to the **LinkOut** pins of other transputers or TRAMs.

LinkSpeedA, LinkSpeedB These select the speeds of **Link0** and **Link1,2,3** respectively. Table 2.3 shows the possible combinations.

LinkSpeedA	LinkSpeedB	Link0	Link1,2,3
0	0	10 Mbits/s	10 Mbits/s
0	1	10 Mbits/s	20 Mbits/s
1	0	20 Mbits/s	10 Mbits/s
1	1	20 Mbits/s	20 Mbits/s

Table 2.3 Link speed selection

ClockIn A 5 MHz input clock for the transputer and CVC. The transputer synthesises its own high frequency clocks. **ClockIn** should have a stability over time and temperature of 200 ppm. **ClockIn** edges should be monotonic within the range 0.8V to 2.0V with a rise/fall time of less than 8 ns.

Reset Resets the transputer, and other circuitry. **Reset** should be asserted for a minimum of 100ms. After **Reset** is deasserted a further 100 ms should elapse before communication is attempted on any link. After this time, the transputer on this TRAM is ready to accept a boot packet on any of its links.

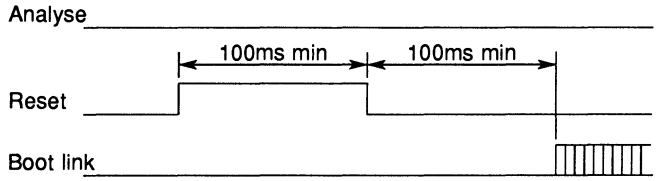


Figure 2.1 Reset timing

Analyse is used, in conjunction with **Reset**, to stop the transputer. It allows internal state to be examined so that the cause of an error may be determined. **Reset** and **Analyse** are used as shown in figure 2.2. A processor in analyse mode can be interrogated on any of its links.

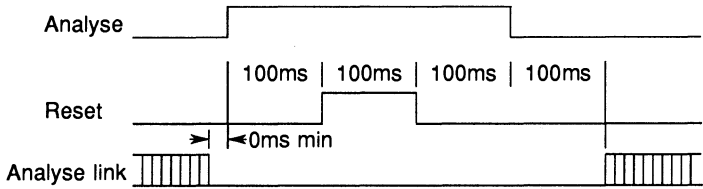


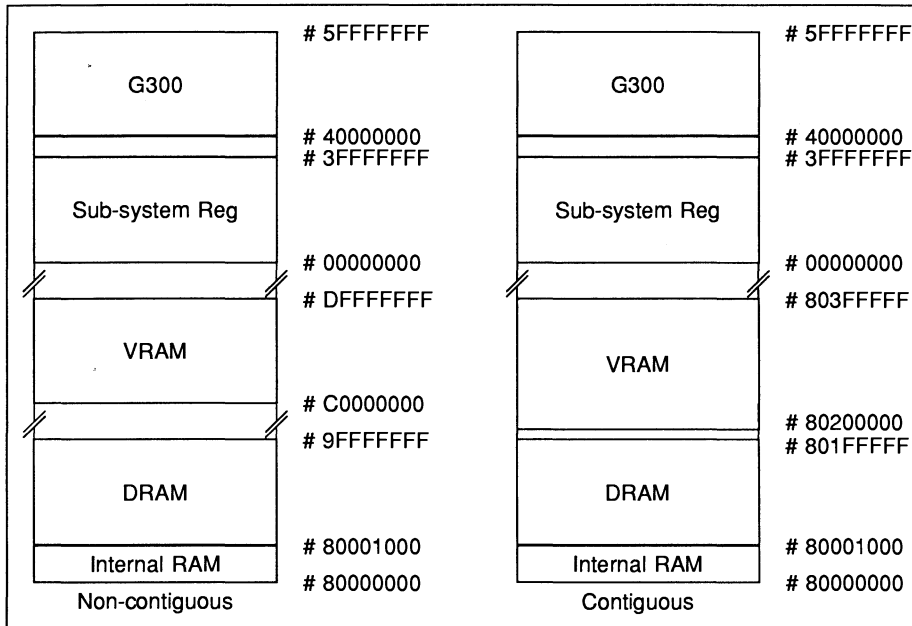
Figure 2.2 Analyse timing

notError An open collector output which is pulled low when the transputer asserts its Error pin. **notError** should be pulled high by a 10kΩ resistor to **VCC**. Up to 10 **notError** signals can be wired together. The combined error signal will be low when any of the contributing signals is low.

2.15.4 Memory Map

The memory space on the board may be divided up into two non-contiguous areas, bitmap and workspace, so that operating systems which use automatic workspace sizing will not trespass on the screen space. A series of PCB links are provided which enables the bitmap and workspace to become contiguous, enabling any spare video RAM to be used as program space if required.

Figure 2.3 shows how the memory is mapped into the address space of the IMS T800 (the “#” sign indicates a hexadecimal number).



Information on the G300 CVCs registers and locations can be found in the *Graphics Databook*, INMOS Limited.

SubSystem registers

The user may require the G300 Graphics TRAM to control a network of transputers and/or other TRAMs. A set of control signals are provided which enables the master to control these slaves or subsystems. The SubSystem port consists of three signals: **SubSystemReset** and **SubSystemAnalyse**, which enables the master to reset and analyse its subsystem; and **SubSystemnotError**, which is used to monitor the error flag in the subsystem.

To maintain software compatibility between TRAMs the SubSystem registers start at hardware address #00000000. These registers are located as shown in table 2.4.

Register	Hardware byte address
SubSystemReset (Wr only)	#00000000
SubSystemAnalyse (Wr only)	#00000004
SubSystemnotError (Rd only)	#00000000

Table 2.4

The SubSystem port operates as follows:

Writing a '1' into bit 0 of #00000000 asserts **SubSystemReset**.
Writing a '0' into bit 0 of #00000000 deasserts **SubSystemReset**.

Writing a '1' into bit 0 of #00000004 asserts **SubSystemAnalyse**.
Writing a '0' into bit 0 of #00000004 deasserts **SubSystemAnalyse**.

A '1' read from bit 0 of #00000000 indicates that **SubSystemnotError** is TRUE.
A '0' read from bit 0 of #00000000 indicates that **SubSystemnotError** is FALSE.

A further two registers are included which enables users to reset the G300 CVC and to switch between the system clock or the on board oscillator. The first of these registers enables users to reset the G300 CVC without resetting the IMS T800, this is important when the application running must not be interrupted. The second register allows users to select a particular pixel dot rate, which may not be attainable using the 5 MHz system clock and PLL multiplication factors. Refer to the G300 clock selection section for further information on input clocks and multiplication factors.

The auxillary control registers operate as follows:

Writing a '1' into bit 0 of #000000F0 asserts G300 CVC reset
Writing a '0' into bit 0 of #000000F0 deasserts G300 CVC reset

Writing a '1' into bit 0 of #000000F4 selects the on board Osc for PLLCikIn
Writing a '0' into bit 0 of #000000F4 selects 5 MHz for PLLCikIn

On power up, or on a system reset the Clock selection register defaults to '0'

2.15.5 IMS G300 clock selection

Alternate clocking schemes are provided which offer a high degree of system flexibility. The primary clocking system utilises the on chip phase-locked loop to multiply the input clock to the full video clock rate.

The second method involves disabling the PII and using a times one clock from the on board oscillator or from an external source.

Table 2.5 shows the recommended input clocks and multiplication factors.

Video data rate (MHz)	PIIClkIn (MHz)	Clock Multiplication
30	6	5
40	6.66	6
50	7.142	7
60	7.5	8
70	7.777	9
80	8	10
90	8.181	11
100	8.333	12
110	8.461	13
120	8.571	14

Table 2.5

The figures shown in the above table are for maximum phase-locked loop stability. The IMS B419 Graphics TRAM uses the system clock (5 MHz) to drive the PII. To achieve a video data rate of 105 MHz a multiplication factor of 21 is used, although these figures are not recommended for use in a noisy environment they provide an extremely stable picture.

If the required video data rate can not be achieved using the system clock and relevant multiplication factor the on board crystal oscillator may be used by writing a '1' into address #000000F4. If the oscillator is used to drive the pixel clock input it must run at the video dot rate with the appropriate jumpers set.

2.15.6 Jumper selection

JP1	Enable PLL	(Jumper removed)
JP2	On board Osc to PixClk in	(JP3 removed)
JP3	External Pix Clock Enable	(JP2 removed)
JP4	Contiguous memory VRAM start	#80200000
JP5	Non-contiguous memory VRAM start	#C0000000

For further information on jumper positions and precautions, please refer to the IMS B419-3 User manual.

2.15.7 Video and sync outputs

The G300 CVC timings comply with both the RS170a and EIA-343 video standard. The outputs are designed to drive a doubly terminated 75R line, thus the effective load seen by the device is 37.5R. The RGB analogue outputs and synchronising signals are brought out to the edge of the board on five SMB connectors as shown below. If the display monitor accepts composite sync on one of its video inputs the sync outputs may be left unconnected.

SMB identification from top to bottom of the board.

1	Pixel clock in	Input (note)
2	Vertical Sync	Output
3	Composite or Horizontal Sync	Output
4	Blue	Output 75R
5	Green	Output 75R
6	Red	Output 75R

Note maximum Pixel clock input is up to 110 MHz, the speed of the G300 CVC.

2.15.8 Mechanical details

Figure 2.4 indicates the vertical dimensions of a single IMS B419 and Figure 2.5 shows the outline drawing of the IMS B419.

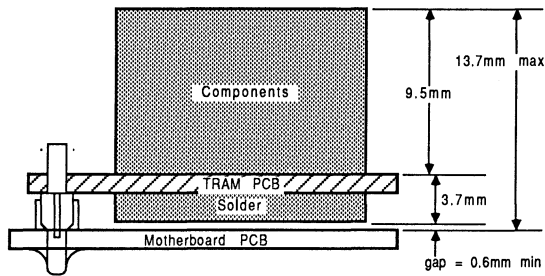


Figure 2.4 IMS B419 height specification

2.15.9 Installation

Since the IMS B419 contains CMOS components, all normal precautions to prevent static damage should be taken.

The IMS B419 may be supplied with spacer pin strips attached to the TRAM pins on the underside of the board. These spacers perform two functions. Firstly, they help to protect the TRAM pins during transit. Secondly, they can be used to space the TRAMs off the motherboard. If there are no components mounted on the motherboard TRAM slot, then the spacer strips should be removed before the TRAM is inserted.

If the subsystem signals are required, plug a 3-way header strip into the solder-side sockets on the IMS B419.

Plug the IMS B419 into the motherboard. Where the IMS B419 is being used with an INMOS motherboard, the yellow triangle marking pin 1 on the IMS B419 (see Figure 2.5) should be aligned with the silk screened triangle that appears in the corner of the appropriate TRAM slot.

Should it be necessary to unplug the IMS B419, it is advised that it is gently levered out while keeping it as flat as possible. As soon as the IMS B419 is removed, the spacer pin strips should be refitted to the TRAM to protect the pins.

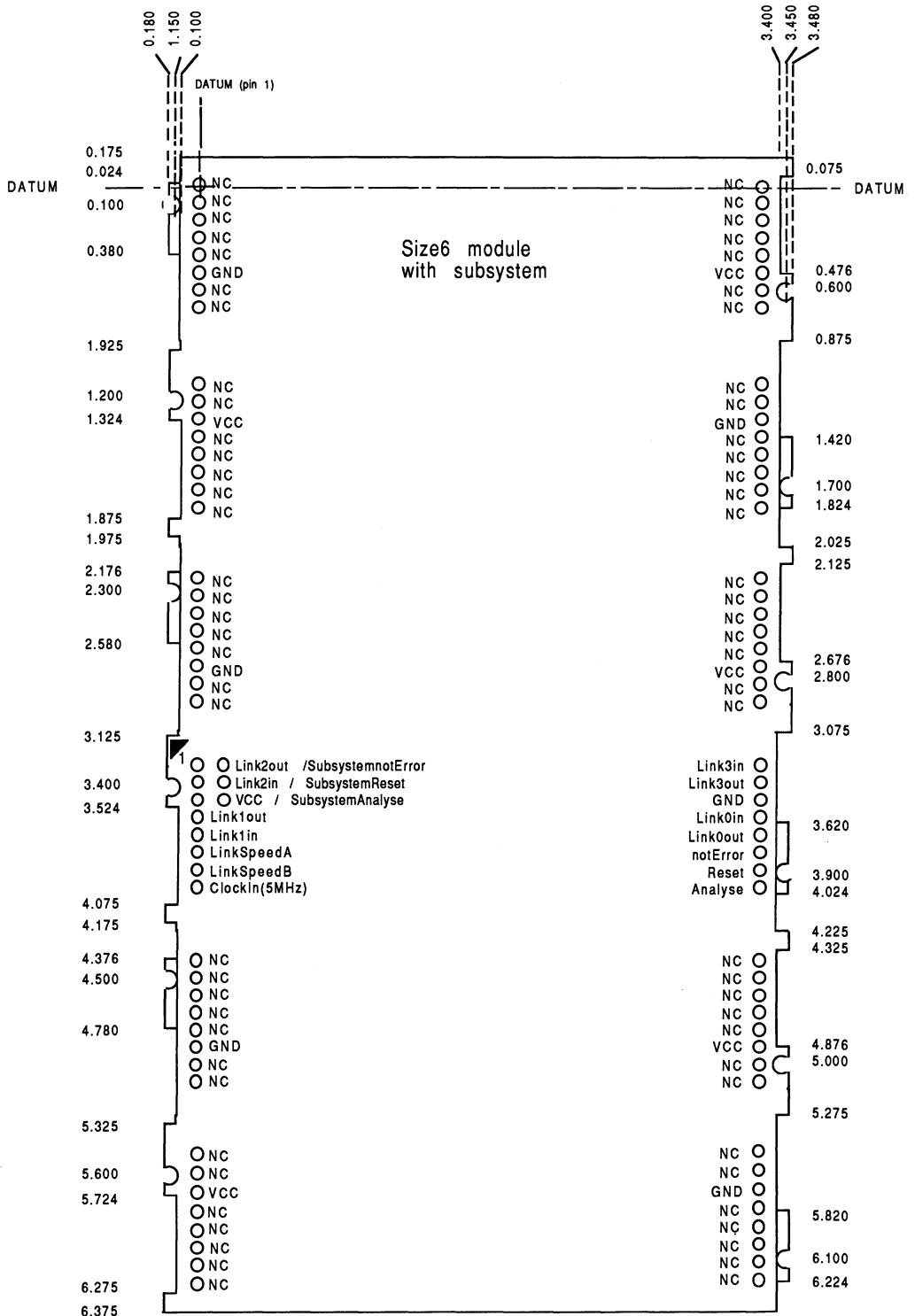


Figure 2.5 IMS B419 outline drawing (All dimensions in inches)

2.15.10 Specification

TRAM feature IMS B419-3		Unit	Notes
Transputer type	IMS T800-20		
Number of transputers	1		
Number of INMOS serial links	4		
Amount of DRAM	2	Mbyte	
Amount of VRAM	2	Mbyte	
DRAM/VRAM cycle time	200	ns	
Subsystem controller	Yes		
Peripheral circuitry	IMS G300-11		
Parity	No		
Size (TRAM size)	6		
Length	3.66	inch	
Pitch between pins	3.30	inch	
Width	6.55	inch	
Component height above PCB	13.7	mm	
Component height below PCB	3.7	mm	1
Weight	175	g	
Storage temperature	0–70	deg C	
Operating temperature	10–40	deg C	2
Power supply voltage (VCC)	4.75–5.25	Volt	
Power consumption	9	W	3

Table 2.6 IMS B419 specification

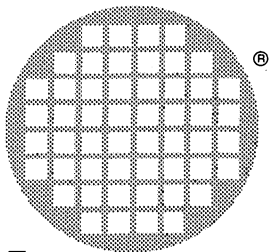
Notes:

- 1 This dimension includes the thickness of the PCB.
- 2 The figure quoted refers to the ambient air temperature.
- 3 The figure quoted has not been characterised and is subject to change.

2.15.11 Ordering Information

Description	Order Number
IMS B419 TRAM with IMS G300-11	IMS B419-3

Table 2.7 Ordering information



inmos

IMS B420

Vector processor TRAM

Size 4

FEATURES

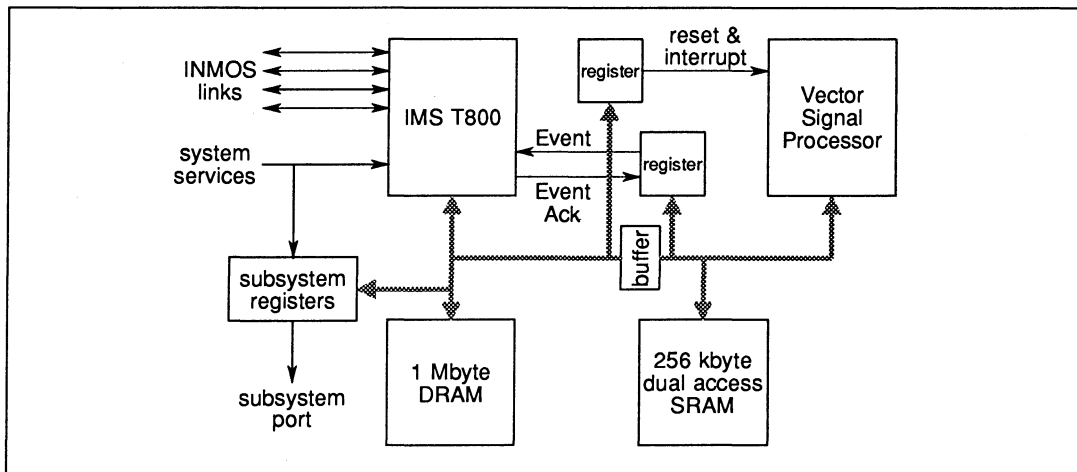
- IMS T800-25 floating point transputer
- High performance vector/signal processing co-processor
– e.g. 1K complex FFT < 2ms
- Both processors support IEEE 754-1985 floating point
- 4 INMOS serial communication links allowing connection of multiple Vec-TRAMs
- 1 Mbyte DRAM for IMS T800
- 256 Kbyte, dual access SRAM for full speed co-processor operation
- Size 4 TRAM
- Sub-system port

GENERAL DESCRIPTION

The IMS B420 VecTRAM is a transputer module combining the communications ability and scalar floating point performance of the IMS T800 with a high performance vector/signal processing co-processor. The two processors can operate concurrently, using separate dynamic and static memory blocks. The vector/signal processor is normally operated as a slave to the IMS T800 which can read and write to the SRAM, to set up vector/DSP routines as well as to load data for processing. The two processors can handshake via interrupts, thus allowing the transputer to initiate vector routines and the co-processor to signal the termination of the requested task.

Examples of the coprocessor's capabilities are: 1K complex FFT in 1.8 ms, 10x10 by 10x10 matrix multiplication in approximately 135 μ s and 64-tap FIR in 6 μ s.

Application areas include speech and image processing, graphics and numerical processing, radar, sonar and seismology.



2.16 IMS B420 VECTRAN product overview

2.16.1 Specification

TRAM feature		Unit	Notes
IMS T800 transputer	1		
ZR34325 vector processor	1		
Fast dual-port RAM	256	Kbyte	
DRAM	1	Mbyte	
TRAM size	4		
Length	3.66	inch	
Width	4.35	inch	
Pitch between pins	3.30	inch	
Component height above PCB	TBA	mm	
Component height below PCB	TBA	mm	1
Weight (approx.)	150	g	
Storage temperature	0–70	°C	
Operating temperature	10–40	°C	
Power supply voltage (VCC)	4.75–5.25	Volt	
Power consumption	TBA	W	

Table 2.1 IMS B420 specification

Notes:

1 This dimension includes the thickness of the PCB.

2.16.2 Ordering Information

Description	Order Number
VECTRAN	B420-1

Table 2.2 Ordering information



Standard Interface Boards

3.1 IMS B008 IBM PC Module Motherboard product overview

3.1.1 Product Overview

The IMS B008 is a full length PC-AT format card which allows transputer systems to interface to the IBM PC-XT or PC-AT bus. It supports up to ten TRAMS plugged into the slots on the board which are configured into a pipeline. An IMS T222 controlling a IMS C004 enables transputer networks to be configured under software control. A connector on the backpanel of the board gives access to links and system services allowing connections to other IMS B008 boards, or to any board compatible with the link and system service signals. The IBM PC bus interface supports DMA and interrupts.

3.1.2 TRAM Slots

The board provides ten TRAM slots which are configured into a pipeline using links 1 and 2 of each TRAM as shown in the block diagram. Jumpers are provided to allow the IMS B008 to be set up either as a head of a pipeline of motherboards or as a board in such a pipeline. Links 0 and 3 from each slot are connected to the IMS C004.

3.1.3 System Services

On all INMOS board products the term "system services" refers to the collection of the reset, analyse, and error signals. On the IMS B008 the system services for the TRAM in slot 0 can be connected to either the UP system services from another board or the system services controlled by the PC bus interface. System services for the other TRAMS can be connected to the same source as TRAM 0 or to the subsystem port of TRAM 0. As shown in the block diagram the Down and Subsystem services are brought out to the 37 way D-type connector allowing this hierarchy to be extended to multi board systems.

3.1.4 Link Configuration

An IMS T222 and a IMS C004 on the board allow the configuring of the TRAMS into different networks under software control. The configuration information is passed to the IMS T222 either by TRAM 0 when the board is at the head of a pipe of boards or from the link ConfigUp from another board. Link 2 from the IMS T222 is taken out to the D-type connector. This allows the IMS T222 devices on all the motherboards in a system to be connected into a pipe allowing configuration information to be passed to each board. The IMS C004 can be hard reset by the IMS T222.

3.1.5 IBM PC Bus Interface

The IMS B008 bus interface is implemented using a IMS C012 link adaptor mapped into the I/O address space of the PC. Five further registers allow software on the PC to control Reset and Analyse signals to the transputer system, read the Error signal from the transputer system, enable interrupts, start and set the direction of a DMA transfer, and select the DMA and interrupt channels to be used. The IMS C012 registers and the interface control registers occupy a 32 byte area in the address map, the base address of which can be located at 150, 200, or 300 (HEX). A memory map is given below.

Memory Map

Board address	Register
boardbase + 00	IMS C012 Input data register
boardbase + 01	IMS C012 Output data register
boardbase + 02	IMS C012 Input status register
boardbase + 03	IMS C012 Output status register
boardbase + 10	Reset register
boardbase + 11	Analyse register
boardbase + 10	Error location
boardbase + 12	DMA request register
boardbase + 13	Interrupt control register
boardbase + 14	DMA and IRQ Channel select register

Interrupts

Interrupts can be generated on the IBM PC bus on any of four interrupt channels 3, 5, 11, or 15; only 3 and 5 being available on the IBM PC-XT. Interrupts can be generated on the following events:

- Data byte received on the IMS C012 link
- IMS C012 ready to send a data byte out on the link
- Transputer system Error signal active
- End of DMA transfer

DMA

DMA logic on the IMS B008 allows data to be transferred between the PC memory and the transputer system at a faster rate than possible using a polling scheme or interrupting on each byte transferred. DMA requests are generated when the IMS C012 is free to transmit a byte or has received a byte depending on the direction of the transfer. These DMA requests can be generated on DMA channels 0, 1, or 3 with only 1 and 3 being available on the IBM PC-XT. Control of the direction of transfer and starting of the DMA process is achieved by writing into the DMA request register.

3.1.6 Link Speeds

The link speeds of the TRAMs, the IMS C012, and the IMS C004 can be set to 10 or 20 Mbits/s. Link speeds for the IMS T222 link 0 can be set 5, 10, or 20 Mbits/s

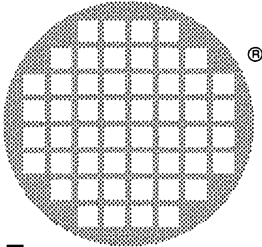
3.1.7 Technical Summary

- IBM PC-XT or PC-AT format card
- 10 TRAM slots
- Reconfigurable using IMS C004 link switch
- IMS T222 for configuration control
- IBM PC bus interface designed around IMS C012
- Bus interface supports interrupts and DMA
- Conforms to the Module Motherboard Architecture

3.1.8 Ordering Information

Description	Order Number
IMS B008 IBM PC Module Motherboard	IMS B008-1

Table 3.1 Ordering information



inmos

IMS B011

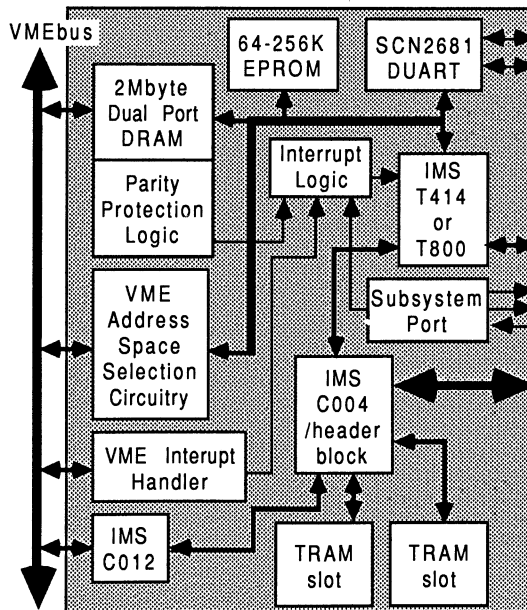
VMEbus master card

FEATURES

- VMEbus system master with interrupt handler
- IMS T800 transputer
- 2 Mbyte dual-ported, parity protected DRAM
- 4 EPROM sites (upto 256 Kbyte)
- 2 RS232 ports
- 2 TRAM slots
- IMS C004 programmable link switch

GENERAL DESCRIPTION

The IMS B011 is a high performance, parallel processing board designed to be integrated into VMEbus based systems. In addition to all the features expected from a VMEbus master card, it performs all the necessary communications between an array of transputers and the VMEbus. In this way the IMS B011 can form a bridge between facilities readily available on the VMEbus (eg. hard disk storage, I/O circuitry) and the processing power of a transputer network (INMOS produces a slave VMEbus board, the IMS B014, with 8 TRAM slots allowing such networks to be implemented very efficiently). Alternatively, when used in conjunction with the IMS D505 SUN-based toolset, the IMS B011 may be used as a transputer development environment operating under UNIX.



3.2 IMS B011 Tranputer VMEbus Master Card product overview

3.2.1 Processor

The INMOS tranputer family of microprocessors is the industry standard in the field of multi-processing. The IMS T800, around which the IMS B011 is based, is a very high performance 32 bit device with floating point unit, 4 Kbytes of single cycle SRAM, and 4 high speed serial communications links all integrated onto the same chip. An interrupt response time of less than 1 μ s makes it particularly applicable to real-time applications.

3.2.2 Booting

A tranputer can be booted either from external ROM or via one of its links. The method of booting the master tranputer on the IMS B011 is selected by a jumper. EPROM sockets are provided (up to 256 Kbytes) for the "boot-from-ROM" option. Alternatively, the tranputer may be booted down a link from the VMEbus interface (via an IMS C012 link adaptor), or direct from a link connection on the P2 edge connector.

3.2.3 Interrupts

The IMS B011 is a full VMEbus interrupt handler, supporting all seven bus interrupt levels. Each of the interrupt levels can be disabled using jumpers. When an interrupt occurs, a tranputer event is generated and the interrupt level is determined by reading a status register. The VMEbus status/ID vector is then obtained by reading the interrupt acknowledge register for that interrupt level.

3.2.4 Memory

The IMS B011 has 2 Mbytes of parity protected DRAM on the board. It is dual-ported so that it is accessible from both the tranputer and from the VMEbus. If a parity error is logged during a memory read from the VMEbus *BERR is asserted. If it is logged during a memory read from the tranputer, the tranputer EVENT input is activated.

The VMEbus appears several times within the transputers memory map. Each occurrence represents a different mode of access to the VMEbus.

The IMS B011 also provides some EPROM sockets (see Booting Section above).

3.2.5 VMEbus Interface

The IMS B011 is designed to be compatible with VMEbus Specification Rev. C.1. The *functional modules* that are incorporated in the IMS B011 are:

- SYSTEM CONTROLLER including
 - SYSTEM CLOCK DRIVER
 - BUS TIMER (BT0(500))
 - ARBITER (PRI)
 - IACK DAISY-CHAIN DRIVER
 - POWER MONITOR
- DTB MASTER (A32/A24/A16; D32/D16/D08(EO))
- DTB SLAVE (A32/16; D32/D16/D08(EO))
- REQUESTER (ROR)

- INTERRUPT HANDLER (IH(1-7))

3.2.6 RS232 ports

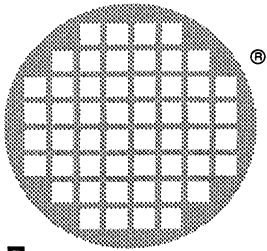
Two asynchronous RS232 ports are implemented using a 2681 DUART. This device has two RS232 ports with handshaking, counter/timer and parallel I/O bits. The counter/timer may be used to generate a periodic event for use in real-time systems.

3.2.7 TRAM slots

The processing power of the IMS B011 may be expanded by adding TRANsputer Modules (TRAMs) into the slots provided. TRAMs are small printed circuit boards containing a transputer and some external memory and they communicate over INMOS serial links using a standard 16 pin interface. Upto 2 of these modules may be fitted to the IMS B011. In addition, an IMS C004 link switch is also provided to allow the interprocessor communication links to be configured under software control.

3.2.8 Ordering Information

Description	Order Number
VMEbus Master with IMS T800	IMS B011-2



inmos

IMS B014

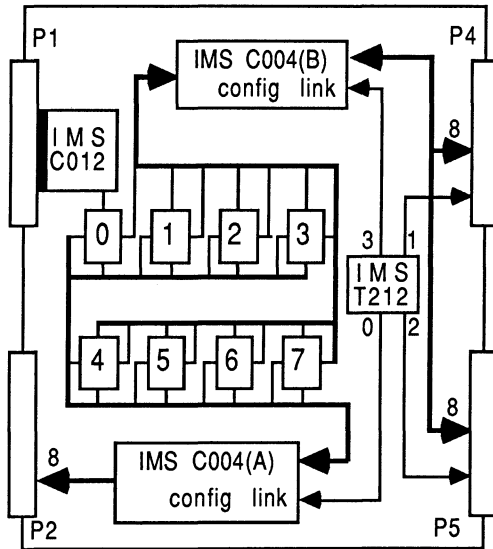
VMEbus slave card

FEATURES

- Compatible with VMEbus Specification Rev. C.1
- Accomodates 8 standard transputer modules (TRAMs)
- Static or dynamic link configuration using two IMS C004 link switches
- Expandable to form arbitrarily large systems
- Suitable for use as VMEbus-transputer interface with IMS D505 SUN based development system

GENERAL DESCRIPTION

The IMS B014 module motherboard is compatible with VMEbus Specification Rev. C.1. It is a standard depth (160mm), double height (6U) card, containing 8 TRAM slots with associated configuration circuitry and a VMEbus slave interface. Two IMS C004 crossbar link switches are provided to allow the user to configure the transputer link connections. This architecture allows any topology to be established on the board. Additionally, 24 links are brought to the edge connectors (8 on the P2 back connector, and 16 split between two front connectors) so that larger networks, using multiple boards, may be constructed.



3.3 IMS B014 VMEbus Module Motherboard product overview

3.3.1 VMEbus Interface

The IMS B014 has a slave interrupting interface to the VMEbus. This interface provides access to a single, bi-directional INMOS link and a system service port. The interface appears as a number of registers located in the A16 (short) address space on the VMEbus, which may be accessed by any VMEbus master such as the IMS B011. These registers are used to program and interact with the IMS B014.

The TRAMS on the IMS B014 can be reset or analysed via the VMEbus interface, or can be bootstrapped through it. Data can be exchanged between TRAMS on the IMS B014 and any bus-master on the VMEbus. All bus communication is achieved using D08(O) data transfers.

3.3.2 Interrupts

The IMS B014 is capable of generating a single VMEbus interrupt that may be assigned to any of the seven VMEbus priority interrupt levels. Interrupts can be triggered by any one of three events:

- data byte received on VMEbus link;
- VMEbus link free to send a data byte;
- an error has occurred in the transputer system;

All interrupts may be individually masked.

3.3.3 IMS C004 Control

The IMS B014 uses the same method of controlling the IMS C004 as other INMOS module motherboards. This allows all IMS C004s to be programmed from a single master configuration link. Each module motherboard has a "config-up" link and a "config-down" link. Thus, motherboards may be cascaded to build multi-board systems, by connecting these links in a pipeline.

On the IMS B014, the "config-up" and "config-down" links can be switched to either the P2 back connector or to the front connectors (P4, P5). Jumpers are also provided that allow either the VMEbus link or slot 0, link 1 to be the master configuration link (figure 3.1).

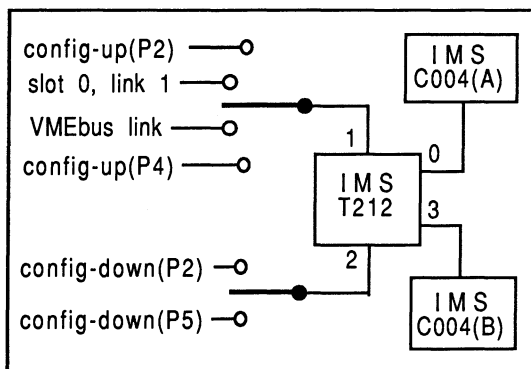


Figure 3.1 Configuration Control

3.3.4 System Services Organisation

On all INMOS board products, the term "system services" refers to the collection of the reset, analyse and error signals. "Reset-up" and "Reset-down" ports are used to carry these signals between boards. Error signals "flow" in the reverse direction to the reset and analyse signals.

The IMS B014 allows system service signals to be generated by bus-masters on the VMEbus. A bus-master can reset or analyse the transputer system by writing to the appropriate registers in the interface. Transputer error signals are propagated back to a register in the interface where they may be monitored by the bus-master.

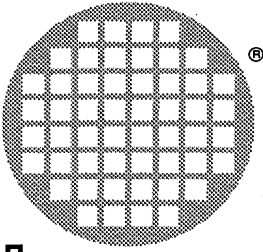
TRAM slot 0 can be reset independently of the other TRAM slots on the board. This allows slot 0 to be used as a "transputer master", controlling other transputers in the system. Thus, it is possible to establish a control hierarchy; a principle that can be extended to multi-level systems using multiple motherboards.

3.3.5 Technical Summary

- 6U (height) by 160mm (depth) VMEbus card
- 8 TRAM slots
- Fully reconfigurable architecture using two IMS C004 link switches
- One IMS T212 transputer for configuration control
- VMEbus interface designed around an IMS C012 link adaptor
- Conforms to VMEbus Specification Rev. C.1

3.3.6 Ordering Information

Description	Order Number
VMEbus Module Motherboard with IMS T212	IMS B014-1



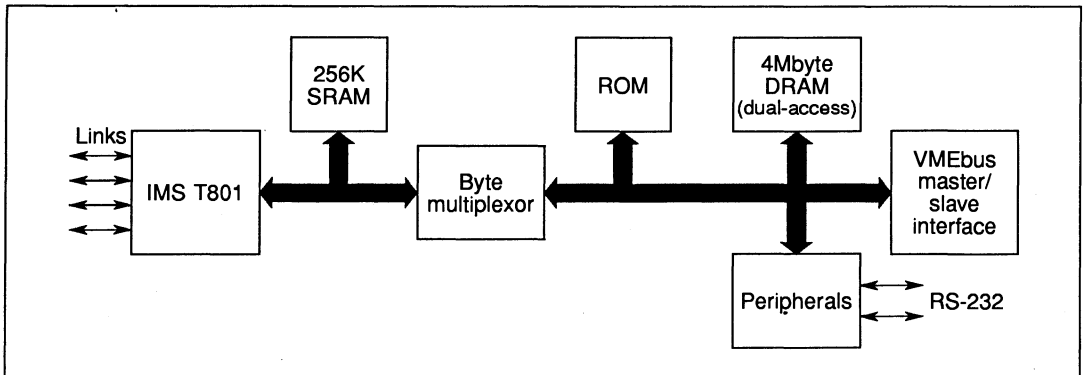
inmos

IMS B016

VMEbus master slave

FEATURES

- VMEbus MASTER/SLAVE.
- IMS T801-25 transputer.
- 256 Kbytes private transputer RAM (80ns cycle).
- 4 Mbytes RAM dual-ported between IMS T801 and VMEbus. Slave DPRAM supports all VMEbus cycle types – A32/24/16, D32/16/8, RMW, UAT, BLT. Write posting supported.
- Full VMEbus master –A32/24/16, D32/16/08, BLT (UAT and RMW are incompatible with IMS T801). Supports all modern request/release modes including fairness requesting.
- Full VMEbus interrupter and interrupt handler.
- Full VMEbus slot 1 functions.
- Databus crosspoint switch between IMS T801 and DPRAM/VMEbus allows fast byte re-ordering to overcome big/little endian incompatibilities.
- 256 Kbytes PROM (in 32-pin JEDEC PLCC sockets).
- 2 serial ports using 2681 DUART.
- Real-time clock for time-of-day, when power supply fails, the RTC is automatically switched to the VMEbus +5V standby rail.
- IMS T801 boots from ROM or link.
- PEX I/O connector for SCSI, Floppy, Parallel, GPIB or custom interfaces.
- Interprocessor communication mailbox registers.
- 6U format board, compliant with VMEbus specification rev C.1.



3.4 IMS B016 VMEbus master/slave Motherboard product overview

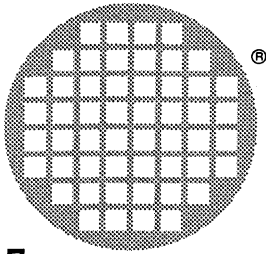
3.4.1 General description

The IMS B016 is a high-performance VMEbus master/slave, based around the IMS T801 32 bit transputer. It is suited to all applications requiring a high performance interface between transputers and the VMEbus. The availability of fast private RAM to the IMS T801 also makes the board suitable for many processing applications.

3.4.2 Ordering Information

Description	Order Number
IMS B016 VMEbus master/slave motherboard	IMS B016-1

Table 3.1 Ordering information



inmos

IMS B015

NEC 9800 series PC Board

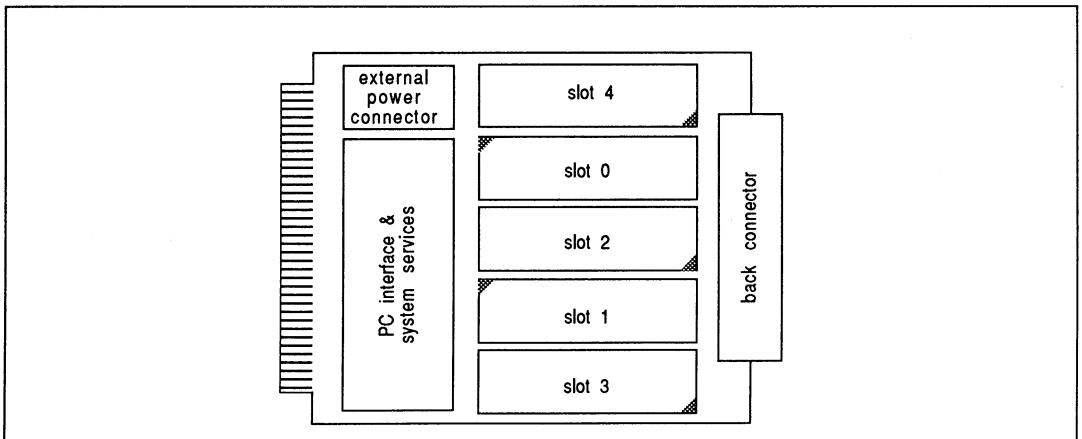
FEATURES

- 5 slots for INMOS TRAMs (Transputer Modules)
- INMOS link adaptor interface to the NEC expansion bus
- Interrupt capability
- Choice of IO address
- Conforms to INMOS module-motherboard architecture
- Can be used as an interface to external transputer systems

GENERAL DESCRIPTION

The IMS B015 is a motherboard for *Transputer Modules* (TRAMs) for the NEC PC-9800 series of personal computers. It allows transputer modules to be fitted to a 9800 series PC for program development, and application acceleration.

The IMS B015 has five slots for TRAMs and an interface to the 9800 series PC expansion bus. This allows the PC to communicate with and reset the TRAMs. It also has connections which allow it to connect to other transputers, TRAMs, or transputer boards (such as another IMS B015).



3.5 IMS B015 Module Motherboard product overview

3.5.1 Link connections

The INMOS link connections on the IMS B015 are arranged as shown in figure 3.1. Two links from each TRAM slot are taken to the back connector (P1). Slots 1-4 are connected in a pipe-line. Some of the link connections can be configured by jumper blocks K1 and K2. K1 connects the PC interface link to SLOT0 link0 or connects both links to P1. K2 connects SLOT0 link2 to SLOT1 link1 or connects both links to P1.

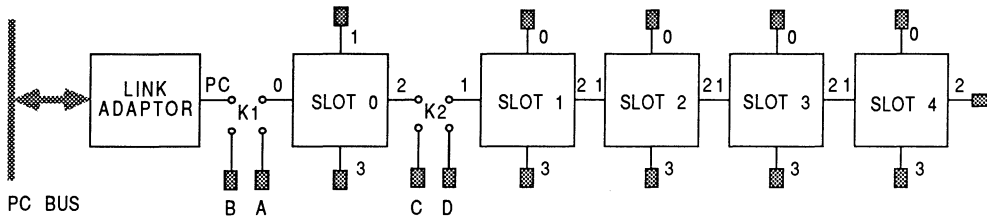


Figure 3.1 link connections

Thus, all of the TRAMs can be connected in a pipeline or all of the links from *slot0* can be taken to P1. The PC interface link and *slot1 link1* can also be taken to P1.

link A in	link A out
slot0 link0 in	slot0 link0 out
PC link out	PC link in
link B out	link B in

Table 3.1 K1 signals

link C out	link C in
slot0 link2 out	slot0 link2 in
slot1 link1 in	slot1 link1 out
link D in	link D out

Table 3.2 K2 signals

3.5.2 Link speed selection

TRAMs have two link speed select pins: **LinkSpeedA** and **LinkSpeedB**. On the IMS B015, there are five link speed select jumpers. These control the TRAM link speeds as shown in table 3.3. To determine the effect of these jumpers on the link speeds of the TRAMs you are using, refer to the data sheets for the TRAMs.

Jumper	Controls	inserted	removed
J9	PC link	10Mbits/s	20Mbits/s
J14	slot0 LinkSpeedA	0	1
J13	slot0 LinkSpeedB	0	1
J11	slot1-4 LinkSpeedA	0	1
J12	slot1-4 LinkSpeedB	0	1

Table 3.3 IMS B015 link speed selection

3.5.3 System Services

A TRAM has a **reset** input, an **analyse** input, and an **notError** output. *System services* is used as a collective term for these signals. The system services signals on the IMS B015 are arranged as shown in figure 3.2.

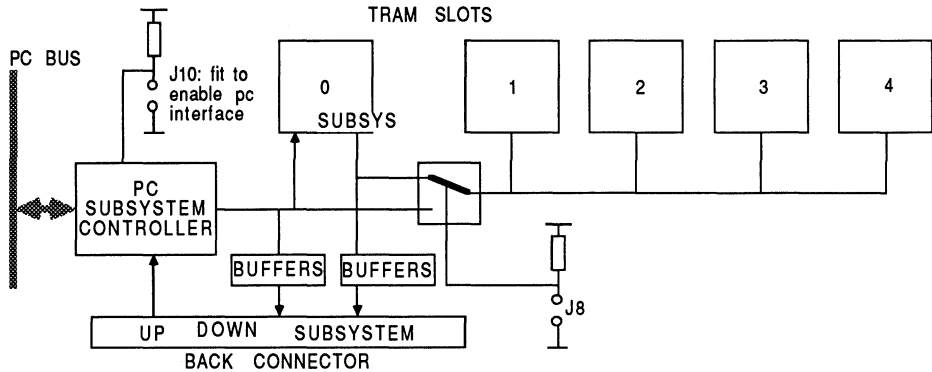


Figure 3.2 System services

If the PC interface is enabled, system services for slot 0 come from the PC interface. If the PC interface is disabled, system services for slot 0 come from the **Up** port.

Some types of TRAM have three extra pins which allow them to drive reset and analyse signals and to monitor an error signal. This is termed *sub-system control*. A TRAM with sub-system control can reset, analyse, and monitor the error signals of other TRAMs. If the TRAM in slot 0 of the IMS B015 has sub-system control, the IMS B015 can be configured so that this TRAM controls the TRAMs in slots 1-4. The sub-system control signals from slot 0 are also available at the **SubSystem** port on the back connector of the IMS B015. This allows the TRAM in slot 0 to control TRAMs on other boards.

TRAMs in slots 1-4 can be controlled by the same system services signals as the TRAM in slot 0; or they can be controlled by the slot 0 sub-system. The selection is made by a single jumper.

3.5.4 Up, Down, and Subsystem

The IMS B015 has three system services ports on the back connector. These are called *Up*, *Down*, and *Subsystem*. Each of these ports comprises a reset, analyse, and error signal.

The Up port allows other TRAMs or transputer boards to control the IMS B015. It can be connected directly to the Down or Subsystem port of another transputer board. The PC interface must be disabled to make use of the **Up** port.

The Down port repeats the reset and analyse signals from the Up port if the PC interface is disabled. If the PC interface is enabled, it repeats the reset and analyse signals from the PC interface. The error signal from the Down port is reported to the PC interface if the PC interface is enabled, and to the Up port if the PC interface is disabled. The Down port can be connected directly to the Up port of another transputer board (such as another IMS B015).

The SubSystem port allows a TRAM with a sub-system controller in slot 0 of the IMS B015 to control other transputer boards. The subsystem port can be connected directly to the Up port of another transputer board.

3.5.5 PC interface

The IMS B015 is designed to plug into any NEC 9800 series PC. It occupies one expansion slot. The interface to the PC expansion bus allows the PC to

- reset the transputer modules
- load code onto the TRAMs
- test the combined error signal from the TRAMs
- analyse the TRAM network to identify the cause of an error
- communicate with the TRAM network

The interface can be polled or interrupt driven.

3.5.6 IO Address

The IMS B015 can be placed at one of several IO addresses in the PC's IO address space. The PC interface can also be disabled so that it does not respond to any address. Table 3.4 shows which jumpers should be fitted to enable the IMS B015 at a particular IO address. The IO addresses are in hexadecimal (indicated by a #). A jumper must be fitted if there is a * in the column, otherwise it must be removed, x means that it does not matter if a jumper is fitted or not fitted. The IMS B015 occupies a block of sixteen addresses starting at the base address.

Base address	J6	J7	J10
#0D0	*	*	*
#1D0	*		*
#2D0		*	*
#3D0			*
none	x	x	

Table 3.4 IO address

3.5.7 Reset, Analyse and Error registers

These registers allow software running on the PC to control the TRAMs on the IMS B015 and to monitor their combined error status. Their offsets from the board base address are given in table 3.5.

Register	Address	Read/Write	Asserted State
reset	base + #8	write only	1
analyse	base + #A	write only	1
error	base + #8	read only	0

Table 3.5 PC system services register locations

Writing 1 to the *reset* register asserts reset to the TRAM in slot 0 and asserts **notDownReset**. Writing 1 to the *analyse* register asserts analyse to the TRAM in slot 0 and asserts **notDownAnalyse**. The *error register* indicates whether any of the TRAMs has asserted its error flag or if **notDownError** is asserted. A 0 is read if any of the TRAMs has asserted its error flag.

3.5.8 Interface link

To allow the PC to load code to the TRAM network, an interface from the PC expansion bus to an INMOS link is provided. The interface uses an IMS C012 link adaptor. This device is like a UART: it has *output data* and *input data* registers, and *output status* and *input status* registers. These are located at the addresses shown in table 3.6.

Register	Address	Read/Write
input data	base + #0	read only
output data	base + #2	write only
input status	base + #4	read/write
output status	base + #6	read/write

Table 3.6 PC interface link register locations

The **output status register** contains an *output ready* bit which is 1 if the output data register is empty. The interrupt enable bit allows the link adaptor to assert one of the NEC PC's interrupt lines when the output data register is empty.

Bit	Name	Function
0	output ready	0 if output data register is busy
1	interrupt enable	1 to enable output interrupt
2-7		none

Table 3.7 Output Status Register

The **input status register** contains a *data present* bit which will be 1 if the input data register contains a valid byte received from the link. The interrupt enable bit allows the link adaptor to assert one of the NEC PC's interrupt lines when the input data register contains data.

Bit	Name	Function
0	data present	1 if data has been received
1	interrupt enable	1 to enable input interrupts
2-7		none

Table 3.8 Input Status Register

Interrupts

The link adaptor can be made to interrupt the NEC PC when it has received or transmitted a byte. Interrupt on output and interrupt on input can be enabled and disabled separately but both conditions drive the same signal to the PC bus. The interrupt signal can be connected to either IR31, IR61 or IR121. These correspond to channels in the PC's programmable interrupt controllers as shown in table 3.9.

Interrupt line	PIC channel	Interrupt Vector	Fit Jumper
IR31	MPIC-IR3	#0B	J5
IR61	MPIC-IR6	#0E	J4
IR121	SPIC-IR4	#14	J3

Table 3.9 Interrupt lines

Only one interrupt line should be driven at any time so only one of J3, J4, J5 should be fitted. If it is not desired to use interrupts, all of the jumpers should be removed.

3.5.9 External power supplies

The NEC PCs can supply a maximum of 0.5A to an expansion board. Because an IMS B015 when populated with TRAMS can require more than this, there is an option to supply power to the IMS B015 from an external power supply while remaining connected to the NEC expansion bus.

The socket for external power (P2) is the same type and pinout as a disk drive power connector. The pinout is given in table 3.10. If you wish to supply power to the IMS B015 from an external supply

- 1 REMOVE J1 and J2.
- 2 Connect a suitable 5V power supply to P2.
- 3 Insert the IMS B015 into the NEC PC.
- 4 Switch on the PC.
- 5 Switch on the power to the IMS B015.
- 6 ALWAYS turn on the IMS B015 power last and turn off the IMS B015 first.

Pin	Signal
1	
2	0V
3	0V
4	5V

Table 3.10 P2 connector pinout

The IMS B015 and any TRAMS will then draw all of their power from the external supply but is interfaced to the PC as before. Note that the 0V of the external power supply is connected to the NEC PC 0V. P2 is also suitable for supplying power to the IMS B015 when it is not connected to a PC.

3.5.10 External Connections

pin	row c	row b	row a
1	GND	GND	GND
2	nc	nc	nc
3	link A out	slot2 link0 out	slot4 link0 out
4	link A in	slot2 link0 in	slot4 link0 in
5	GND	GND	GND
6	GND	GND	GND
7	nc	nc	nc
8	slot0 link1 out	slot2 link3 out	slot4 link3 out
9	slot0 link1 in	slot2 link3 in	slot4 link3 in
10	GND	GND	GND
11	GND	GND	GND
12	nc	nc	nc
13	link C out	slot1 link0 out	slot3 link0 out
14	link C in	slot1 link0 in	slot3 link0 in
15	GND	GND	GND
16	GND	GND	GND
17	nc	nc	nc
18	slot0 link3 out	slot1 link3 out	slot3 link3 out
19	slot0 link3 in	slot1 link3 in	slot3 link3 in
20	GND	GND	GND
21	GND	GND	GND
22	nc	nc	nc
23	link B out	link D out	slot4 link2 out
24	link B in	link D in	slot4 link2 in
25	GND	GND	GND
26	nc	nc	nc
27	nc	nc	nc
28	notUpReset	notSubsystemReset	notDownReset
29	notUpAnalyse	notSubsystemAnalyse	notDownAnalyse
30	notUpError	notSubsystemError	notDownError
31	GND	GND	GND
32	GND	GND	GND

Table 3.11 IMS B015 back connector pinout

3.5.11 Specification

	feature	Unit	Notes
TRAM slots	5		
Interface type	IMS C012 link adaptor to NEC PC expansion bus		
Length	6.65	inch	
Width	5.85	inch	
Component height above PCB	12.0	mm	
Component height below PCB	4.0	mm	1
Weight	150	g	
Storage temperature	0–70	°C	
Operating temperature	10–40	°C	
Power supply voltage (Vcc)	4.75–5.25	Volt	
Power consumption	1.7(typ.) 2.1(max.)	W	

Table 3.12 IMS B015 specification

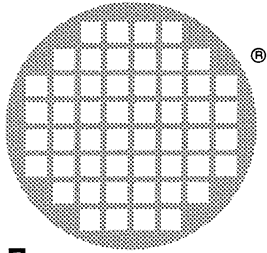
Notes:

1 This dimension includes the PCB thickness of 1.6mm.

3.5.12 Ordering Information

Description	Order Number
IMS B015 Module Motherboard	IMS B015-1

Table 3.13 Ordering information



inmos

IMS B012

Double extended eurocard

FEATURES

- 16 transputer module (TRAM) slots
- IMS T212 - 16 bit transputer
- Two IMS C004 programmable 32 way switches
- Double Extended Eurocard
- 40 links available at edge connectors

GENERAL DESCRIPTION

The IMS B012 is a eurocard TRAM motherboard designed to fit into standard card cages for double extended eurocards (the INMOS ITEM provides such a card cage complete with power supplies, and connectors). The IMS B012 provides 16 slots for TRAMs, with IMS C004s to provide a wide variety of configurations. A possible application might be an image or speech recognition system using eight IMS B401s for feature extraction and a IMS B405 running LISP or another AI language for recognition.

The IMS B012 can also be used to provide switchable backplane connectors for other boards plugged into a backplane.

3.6 IMS B012 Double Eurocard Motherboard engineering data

3.6.1 Introduction

The IMS B012 is a eurocard TRAM motherboard which is designed to fit into standard 6U, 220mm deep, DIN41494 (and IEC 297) card cages such as the INMOS ITEM. It has slots for up to 16 TRAMs. These are transputer-based circuit modules which communicate with the outside world by means of INMOS serial links (a link is a two-wire serial communications port which can run at up to 20 MHz).

The smallest TRAM is 'size 1'. Each of the 16 sites for modules on the IMS B012 will accept a size 1 module. Each module site, or 'slot' has connections for four INMOS links which are designated *link 0*, *link 1*, *link 2* and *link 3*. TRAMs which are larger than size 1 can be mounted on the B012. A larger module occupies more than one slot and need not use all of the available link connections provided by the slots which it occupies.

The B012 has two IMS C004 link switch ICs. These devices are able to connect together links from the slots and 32 links which are available on an edge connector. The connections can be changed by control data passed to the board down a configuration link, which may come from some master system or from one of the TRAMs on the B012 itself.

3.6.2 Hardware Description

The 16 module sites or slots provided by the IMS B012 are 16-pin sockets in accordance with the *TRAM Specification* (INMOS Technical Note 29). The slots are numbered as shown on the board silk screen and in figure 3.1.

The IMS B012 has two DIN41612 96-way edge connectors, P1 and P2. These carry almost all signals and power to/from the board and are easily identified from the board silk screen printing and from figure 3.1. P2 carries power, pipeline and configuration links and system control signals (reset and analyse and error).

NOTE - it is very important that you do not mix up P1 and P2. Unrecoverable damage to the IMS B012 will almost certainly result.

Link Connections

The link connections to the 16 slots are organised as follows:

Two links from each slot (links 1 and 2) are used to connect the 16 slots as a 16-stage pipeline (in a pipeline, multiple processors are connected end-to-end as in figure 3.2). The pipeline is actually broken by jumper block K1. K1 will usually be jumpered in the standard way to give a 16-stage pipeline but can allow other combinations.

When modules larger than size 1 are used, the pipeline will be broken at the slots which are underneath large modules. Special plugs, called pipe-jumpers are provided (figure 3.3 shows a pipe jumper). These plug into the unused slot and connect the signals for links 1 and 2 together, thus connecting the pipeline through to the next TRAM in the chain.

Link 1 on slot 0 is wired to an edge connector (P2) and is called *PipeHead*. Link 2 on slot 15 is also taken to P2 and is called *PipeTail*. By connecting the pipe heads and tails from multiple boards together, a large, multi-board pipeline is created.

The other two links (links 0 and 3) of each slot are, in general, connected to two IMS C004 programmable link switches (For detailed information on the IMS C004 see *The Transputer Databook*).

The IMS C004 has 32 input pins and 32 output pins, plus an INMOS link (ConfigLink) used to send configuration information to the IMS C004. Any of the output pins can be 'connected' to any of the input pins, so a signal presented on the input pin would be buffered and transmitted on the output pin (with a slight delay).

The switch connections are made according to information sent to the IMS C004 down its ConfigLink. The two IMS C004s on the IMS B012 allow 64 link connections to be made under software control.

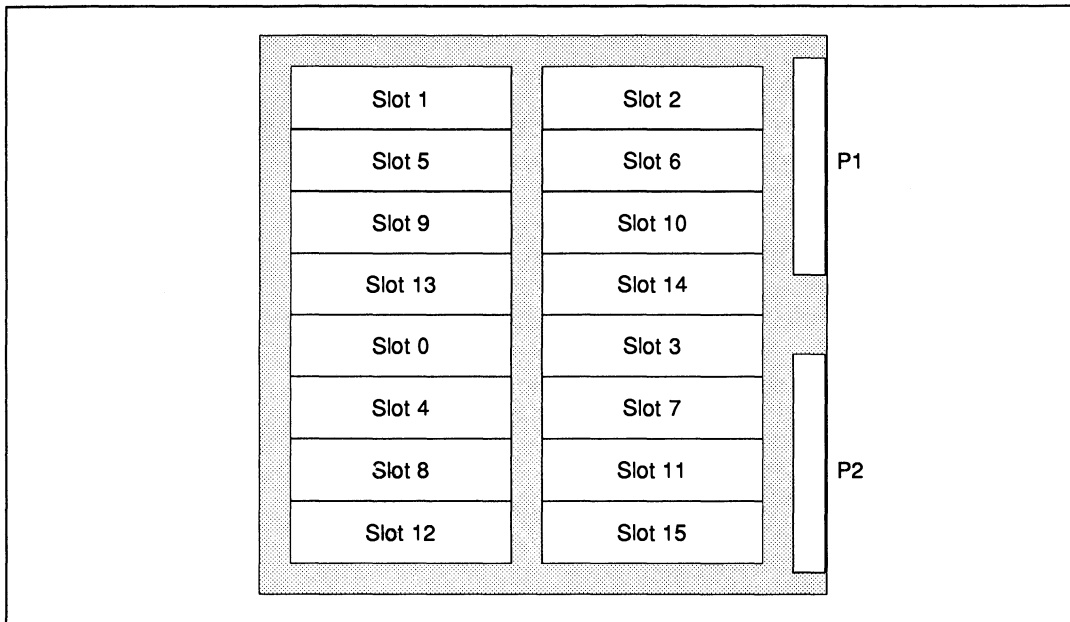


Figure 3.1 IMS B012 slot positions

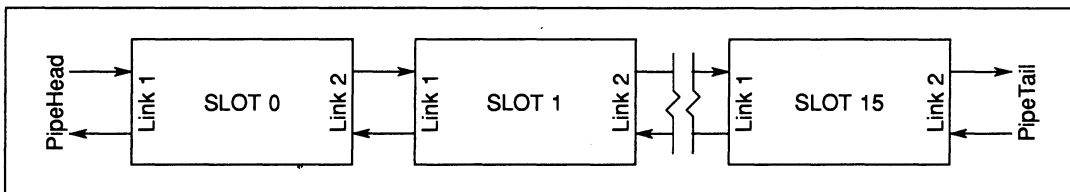


Figure 3.2 A module pipeline

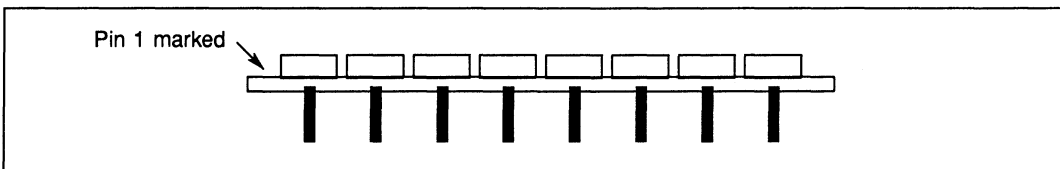


Figure 3.3 A Pipe-Jumper

In most applications using the IMS C004, the device is treated as a 32-way Link Crossbar. This means that 32 INMOS links, each of which has two signals, may be connected to each other in an arbitrary fashion. That is to say that any of the 32 links can be 'connected' via the IMS C004 to any of the other 31 links. The IMS B012 uses the IMS C004s in a slightly different way, the difference being that the two signals from any particular link are routed through *different* IMS C004 devices. So if the LinkIn signal comes from one IMS C004, then the LinkOut signal will go to the other IMS C004. Figure 3.4 shows the general routing of link signals.

The link output signals from all the link 0s on all the slots (16 signals) are connected to 16 inputs of one IMS C004 (IC2). The link input signals from all the link 3s on all the slots (16 signals) are connected to 16

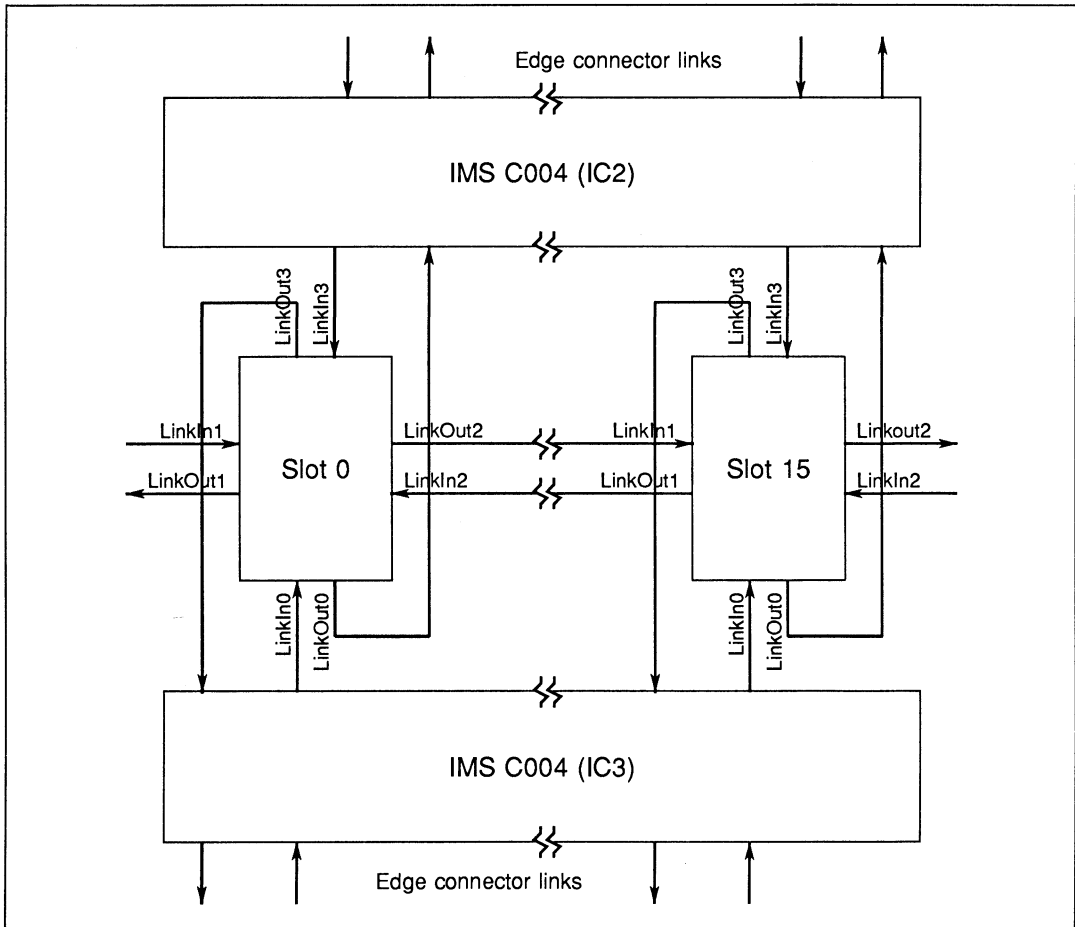


Figure 3.4 Link Organisation - slots to IMS C004s

outputs of the same IMS C004. The remaining 16 inputs and 16 outputs of that IMS C004 are connected to an edge connector (P1).

The other IMS C004 (IC3) is connected similarly, except that 16 of its inputs are connected to the outputs of all link 3s on all the slots, and 16 of its outputs are connected to the inputs of all link 0s on all the slots. The remaining inputs and outputs are connected to P1.

The result of this connection scheme is that any link 0 on any module may be routed via the IMS C004s to *any* link 3 on any module, but may not be routed to *any* of the link 0s on any other module. The same is true for link 3s on any modules, they may not be routed to any other link 3. Each of the links 0 and 3 on any module may be routed to any of half of the link connections on edge connector P1 (see below).

By hardwiring two of the edge connector links together off the board, any of the slot link 0s can be routed to another slot link 0, via the two connected edge links.

MMS (Module Motherboard System) software which is available for all module motherboards allows the configuration of module interconnection to be achieved easily from a connection list description of the desired network.

Slot 0 link 0 is not directly connected to its appropriate IMS C004 pins. It is connected to edge connector P2, along with the respective pins from the IMS C004s. A link jumper connector which is supplied with the board can be used to make the connection between slot 0 link 0 and the IMS C004s. slot 0 link 0 is taken to P2 in order to provide two links which are directly connected to module 0 on an edge connector. For some applications it will be useful to by-pass the IMS C004 switches in this way.

Similarly slot 0 link 3 is connected to pins on jumper-block K1. Usually K1 will be configured to connect slot 0 link 3 to the appropriate pins on the two IMS C004s. Because there are K1 pins which are connected to pins on edge connector P2, slot 0 link 3 can be wired to the edge connector instead of to the IMS C004s.

It is possible, using a non-standard configuration of K1, to take links 0, 1 and 3 from slot 0 off the board via P2. This is useful if slot 0 contains a TRAM which is controlling a system of other TRAMs or transputers.

Figure 3.5 shows the organisation of the pipeline links and the links which are available on P2 and K1.

IMS C004 link switches introduce a small delay into the signals which they switch. If multiple IMS C004s are introduced into a link signal path, as with multi-board IMS B012 systems, the link data rate may be reduced.

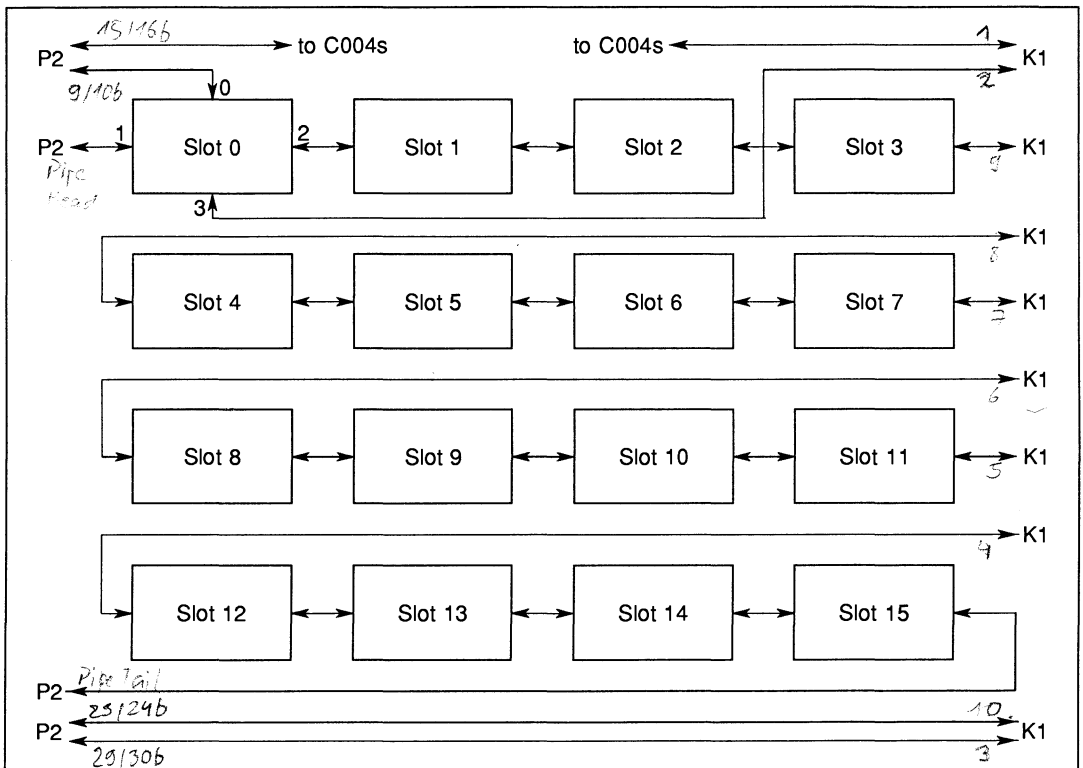


Figure 3.5 Links Available on P2 and K1

P1 Links

Connector P1 has three rows of 32 pins. All the pins in row 'a' are connected to ground. All the pins in row 'b' are link *inputs* and all the pins in row 'c' are link *outputs*. At each of the 32 positions along P1, the three pins from rows a, b and c together carry one link. These signals may be connected to devices with link ports in any way the user desires, as long as the correct electrical precautions required when dealing with links are taken into account. (see section 3.6.2).

A special connector with a small PCB attached and press-fit pins fitted into this PCB is supplied with the IMS B012. This 'mini-backplane', when fitted to P1, allows standard INMOS link cables to be plugged into P1 links. Note that these link cables are not designed for use in arduous physical environments (vibration, corrosion and pulling on the cable). This is why the IMS B012 is provided with standard DIN 41612 connectors. The user may design a connection method for attaching signals to the board which best suits the particular application.

The 32 links available on edge connector P1 are numbered, for reference, starting at 'edge link 0' on pins 1 (a,b and c) through to 'edge link 31' on pins 32 (a,b and c) (see figure 3.6). This numbering scheme is for convenience and there is no obvious mapping between these numbers (the order on the edge connector) and the links to which they are connected on the IMS C004s.

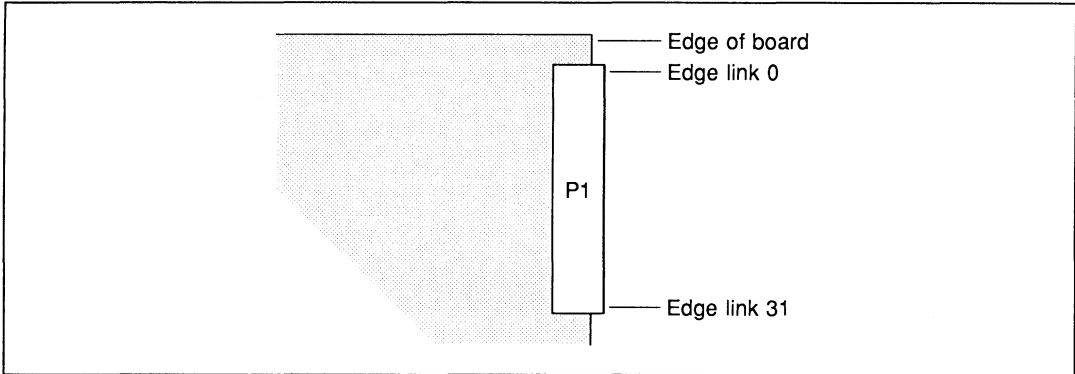


Figure 3.6 P1 Connections

As explained in section 3.6.2, the IMS B012 link switching organisation, using the two IMS C004s, does not allow complete freedom to connect any link to any other link. The following table shows which P1 edge connector links (numbered as above) may be connected to which links on the slots (via the IMS C004 link switches):

P1 Edge Link	To TRAM slot links	P1 Edge Link	To TRAM slot links
0	3	16	3
1	3	17	0
2	0	18	0
3	0	19	3
4	0	20	0
5	0	21	0
6	3	22	3
7	3	23	3
8	3	24	3
9	3	25	3
10	0	26	3
11	0	27	3
12	3	28	0
13	0	29	0
14	0	30	0
15	3	31	0

The link connections on connector P1 are intended mainly for communication between the IMS B012 and other boards. However, it is also possible to use these P1 links and the IMS C004 link switches to switch link

connections for an external system. For instance, two IMS B012 boards in a card cage, unpopulated with TRAMs, may act as a 'programmable backplane' to other boards in the card cage. The connections between these boards and the IMS B012s being hard-wired.

Switch Configuration Transputer

The IMS C004 devices are controlled by an IMS T212 16-bit transputer. The IMS T212 has four links. Links 0 and 3 are connected to the two IMS C004s (link 0 to IC2 and link 3 to IC3). Link 1 is available on edge connector P2 and is called *ConfigUp*. Link 2 is also available on P2 and is called *ConfigDown*. The organisation of these links is shown in figure 3.7.

Configuration data for the IMS C004 is fed into one of the IMS T212's links (*ConfigUp*) from the master configuration system which must be connected to P2. The configuration system could be one of the TRAMs on the IMS B012, provided that one of its links may be connected to *ConfigUp*.

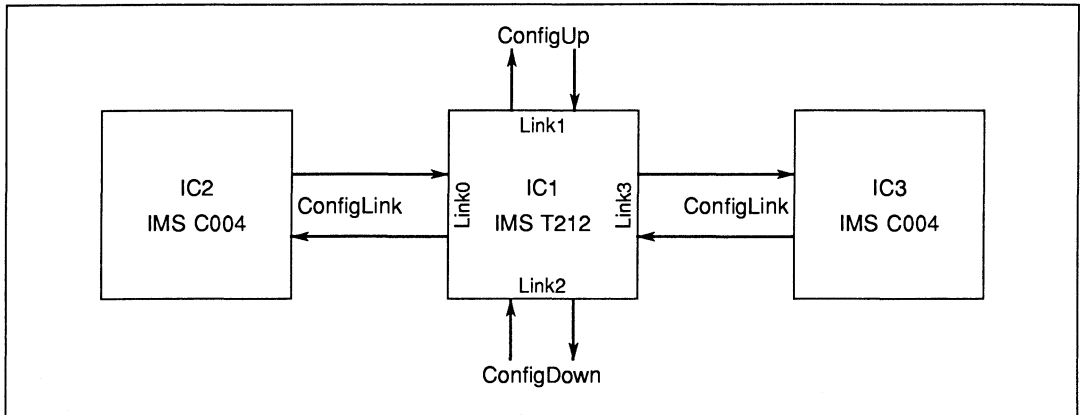


Figure 3.7 Link Configuration Organisation

The IMS T212 has 2 Kbytes of on-chip RAM. A transputer such as the IMS T212, after it is reset, is able to treat bytes sent to it down any of its links as 'boot code'. This is how the IMS T212 on the IMS B012 is used. The configuration host system sends a small program to the IMS T212. The code is stored in on-chip RAM and takes information from *ConfigUp*, routes data out to the two IMS C004s and sends on data out of *ConfigDown* (if required). The MMS (Module Motherboard System) handles all these software functions transparently.

In a multiple motherboard system it is intended that the *ConfigUp* and *ConfigDown* links from adjacent boards be connected together to form a configuration daisy-chain (see figure 3.8). Again the MMS is able to handle the configuration of multiple-board systems (see the *MMS User Manual*).

This configuration architecture is fully compatible with other INMOS TRAM motherboards. Multiple board systems may contain different motherboard types. For instance, an IMS B008 fitted to an IBM PC/XT or PC/AT or compatible, may be part of a system containing multiple IMS B012 motherboards fitted into a card cage.

Reset, Analyse and Error

The Reset, Analyse and Error pins of TRAMs (and transputers) will be referred to collectively as 'system services' in this section. The system service signals are used to reset TRAMs and transputers, to place transputers in an analyse state (for debugging) and to carry the fact that an error has occurred in one processor in an array back to some host system which will deal with the error condition.

Some TRAMs and most evaluation boards are capable of generating the system services for other TRAMs and transputers. This is called a 'subsystem' control capability. The IMS B012 can be connected to another board with subsystem control and can also accommodate one TRAM with subsystem control. Furthermore, the

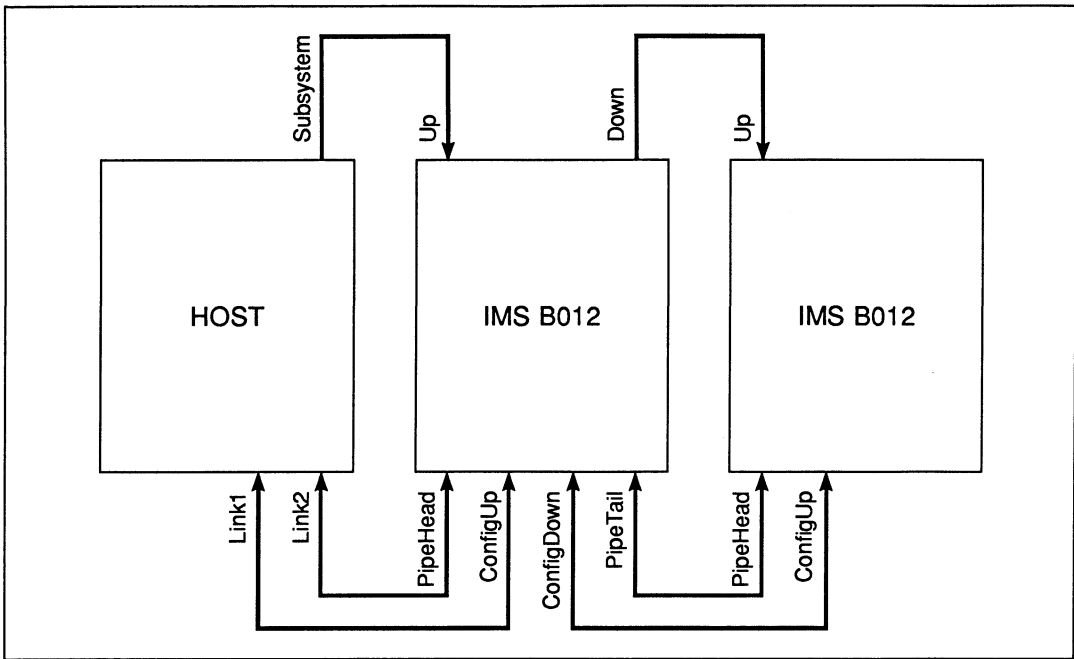


Figure 3.8 Multiple-Board Daisy-Chain

IMS B012 can generate subsystem control signals for other boards.

System services for TRAMs are slightly different to system services for boards since the Reset and Analyse signals are active low for boards and active high for TRAMs.

The TRAMs and other circuitry on the IMS B012 splits into two sections for system services. System services for slot 0 come from the 'Up' pins on edge connector P2. System services for slots 1 to 15 and IC1 (the IMS T212) can either come from from 'Up' as slot 0, or from the Subsystem pins of slot 0, depending upon the state of switch 6.

The IMS T212 Error pin is unconnected so an error condition on IC1 can not propagate into the TRAM array.

Note that slot 0 is the only slot which has subsystem pins and that in order to use these pins it is necessary to have a module with subsystem capability installed in slot 0.

The system service signals for slot 0 are buffered and output on edge connector P2 as the 'Down' pins. This allows system services for multiple boards to be daisy-chained, the 'Down' of one board being connected to the 'Up' of the next.

Figure 3.9 shows the complete organisation of the system services (reset, analyse and error) on the board.

Reset and Analyse signals presented to the IMS B012 on connector P2 should have minimum low pulse widths of 1 millisecond. The subsystem pins of slot 0 are also buffered and are available on edge connector P2 as the 'Subsystem' pins.

The two IMS C004 link switches have a reset pin that is driven by a power-on-reset circuit. The IMS C004s can be also soft-reset by a command from the IMS T212.

The IMS T212 has 2 Kbytes of on-chip RAM. It also has an external memory interface. Circuitry on the

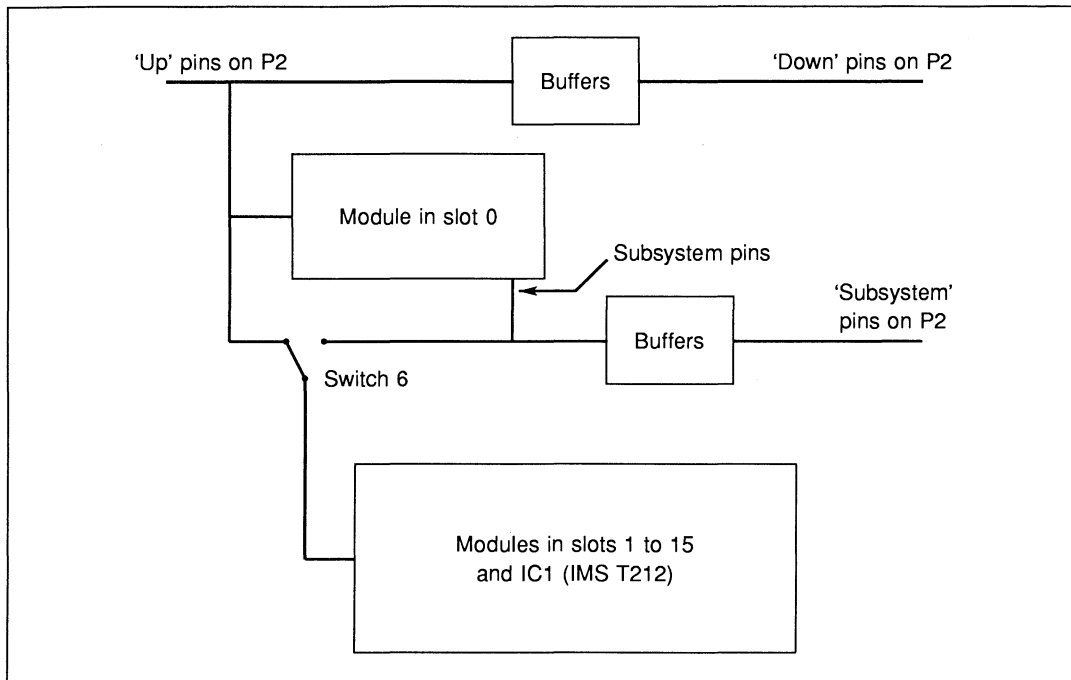


Figure 3.9 IMS B012 System Services Organisation

IMS B012 is connected to the IMS T212's external memory interface which allows the reset signal to the IMS C004s to be controlled from the IMS T212. By writing a *one* into bit position *zero* in any external memory word, the reset signal to the IMS C004s is *asserted*. Similarly, by writing a *zero* into bit position *zero* in any external memory word, the reset signal to the IMS C004s is *de-asserted*.

Note that if the IMS T212 (IC1) writes to external memory and sets the IMS C004 reset signal, subsequent resetting of the IMS T212 will not alter the level of the IMS C004 reset signal. Note also that if the IMS T212 reads from any location in its external memory space then the IMS C004 reset signal will be set to an unpredictable level.

Link Termination

INMOS serial links have two signals, linkIn and linkOut. If a link is to be connected over a distance or between boards then some extra discrete components are needed. The linkOUT signal must be series terminated to match its load, and the linkIn should have a diode to VCC for ESD protection and a pulldown resistor to prevent the receiving transputer booting itself from a floating linkIn (see figure 3.10). The whole question of link connections is covered in detail in INMOS *Technical Note 18*.

- 1 All links on TRAMs have the link termination and protection components on the module. The PipeHead and PipeTail link connections from P2 are directly connected to the module pins (and are therefore terminated). Link 0 from slot 0 is also directly connected to P2.
- 2 The ConfigUp and ConfigDown link connections to the IMS T212 are connected to P2 via termination and protection components.
- 3 All the links on P1 (from the IMS C004s) are terminated and protected.

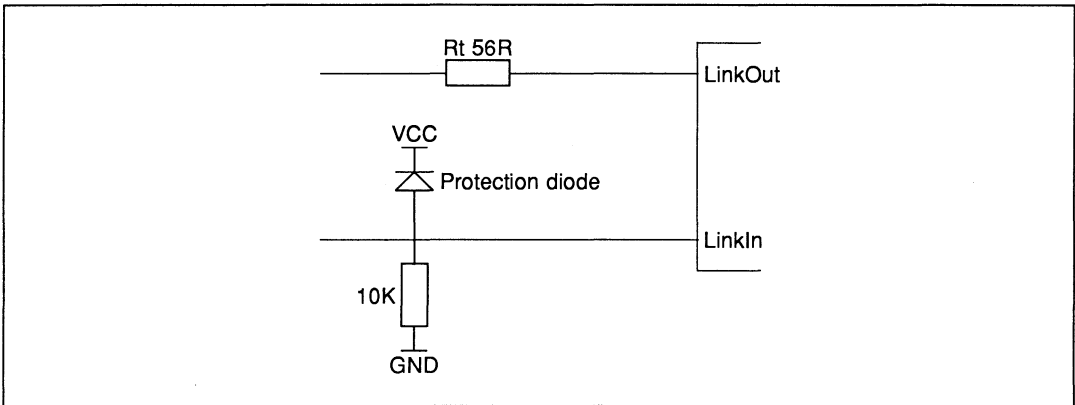


Figure 3.10 Link Termination and Protection

- 4 The link signals from the IMS C004s which would usually be connected to link 0 of slot 0 (via the jumper connector on P2) are terminated and protected.
- 5 The link signals from the IMS C004s which would usually be connected to slot 0, link 3 (via K1) are terminated and protected since it is possible to route these signals directly to an edge connector (P2).

This means that any link available on the edge connector of an IMS B012 is correctly terminated and protected.

Error Lights

Three yellow LED indicators are mounted on the edge of the board, opposite P1 (see figure 3.11). An indicator will be lit when a module asserts its error pin. One LED, LD1, monitors error from slot 0. The other two LEDs, LD2 and LD3, monitor error from the modules on the front row (not including slot 0), and back row of slots respectively. The front row is the group of seven slots situated along the front-panel side of the board (not including slot 0). The back row is the group of eight slots situated along the opposite edge of the board (see figure 3.11).

When the IMS B012 is installed in a card cage, LD1 is the lower of the LEDs; LD2 is the middle one and LD3 is the upper LED.

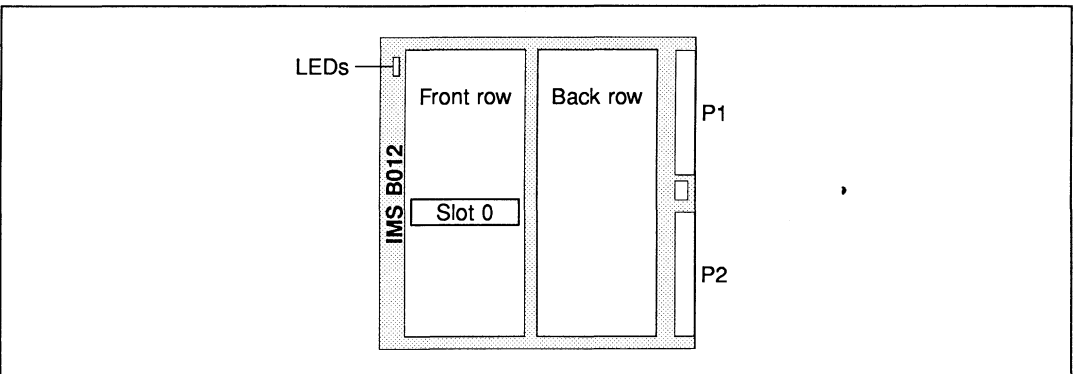


Figure 3.11

User Power Connector

A four pin power connector (designation P3) is mounted near the front edge of the board as shown in figure 3.12. P3 is wired to 0v, +5V and via a wide PCB track to 2 pins on P2. This connector type is the kind used on most floppy-disk drives and when the appropriate pins on P2 are wired to +12V, P3 may be used to power disk drives or similar equipment. Users may take other power signals to P3, such as ECL power supplies.

Pin 4 is connected to connected to 5V, pins 3 and 2 are connected to 0v, pin 1 is pins 3a and .3c on P2. Pin 4 is the top pin when the board is viewed as in figure 3.12.

These power pins can carry up to 3A of current and pin 1 can have up to 50V with respect to GND.

There is a pin post fitted in one corner of the board (marked GND on the silk screen). This is connected directly to the 0V plane and can be useful for attaching 'scope probe ground leads.

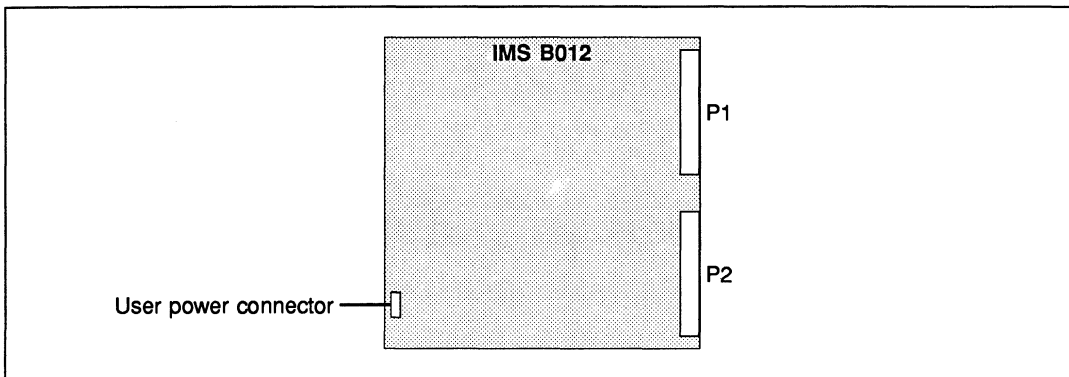


Figure 3.12 P3 Position

Uncommitted Pins

Nine pins on connector P2 are brought to a row of pads near to the connector at the edge of the board (see figures 3.13 and 3.14). These pads, designated P4, may be hard-wired into any circuitry on the B012 by the user for special applications. Possible uses would be RS-232 serial lines, analog signals and extra subsystem control signals. The remaining two pads of P4 are connected to ground. Pin posts can be inserted into the holes which make up P4. The posts could take a single-in-line connector similar to those used in the IMS B012 cable set.

These pins should carry no more than 50mA of current at no more than 25V with respect to GND. Note that these signals, although they are short, have not been designed to carry analog signals and may be susceptible to crosstalk.

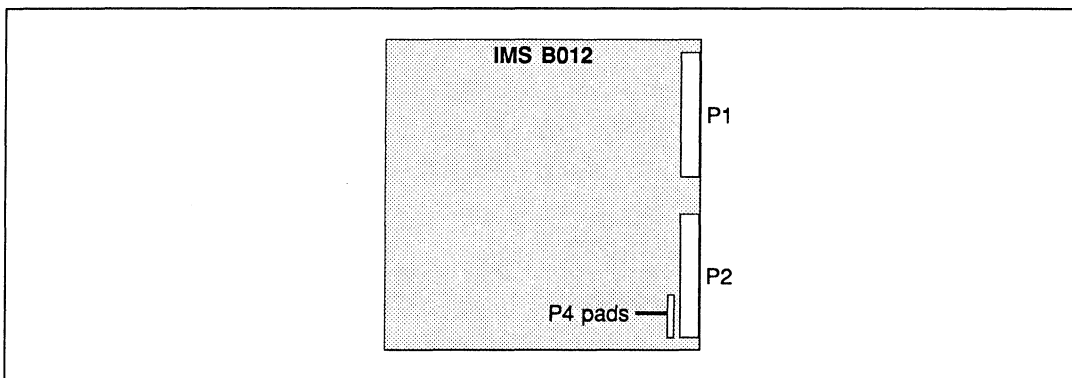


Figure 3.13

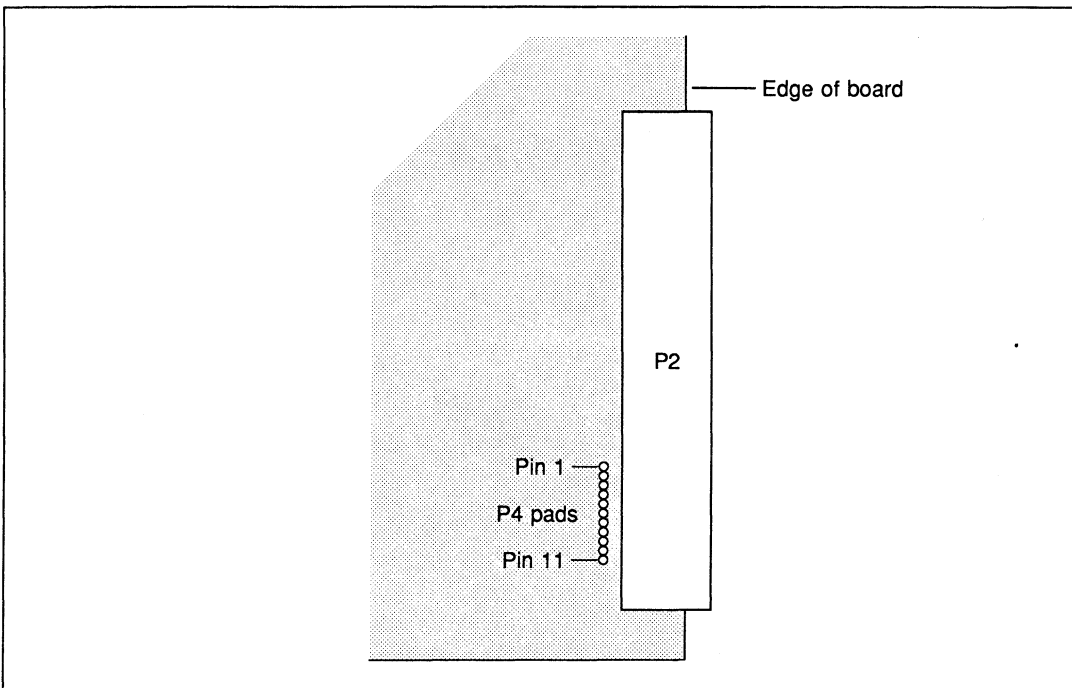


Figure 3.14

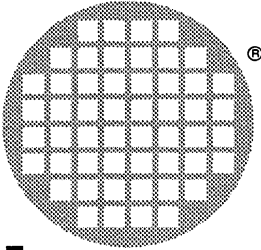
3.6.3 Ordering Information

Description	Order Number
IMS B012 Double Eurocard Motherboard	IMS B012-1

Table 3.1 Ordering information



Evaluation Boards



inmos

IMS B005

Disk board

FEATURES

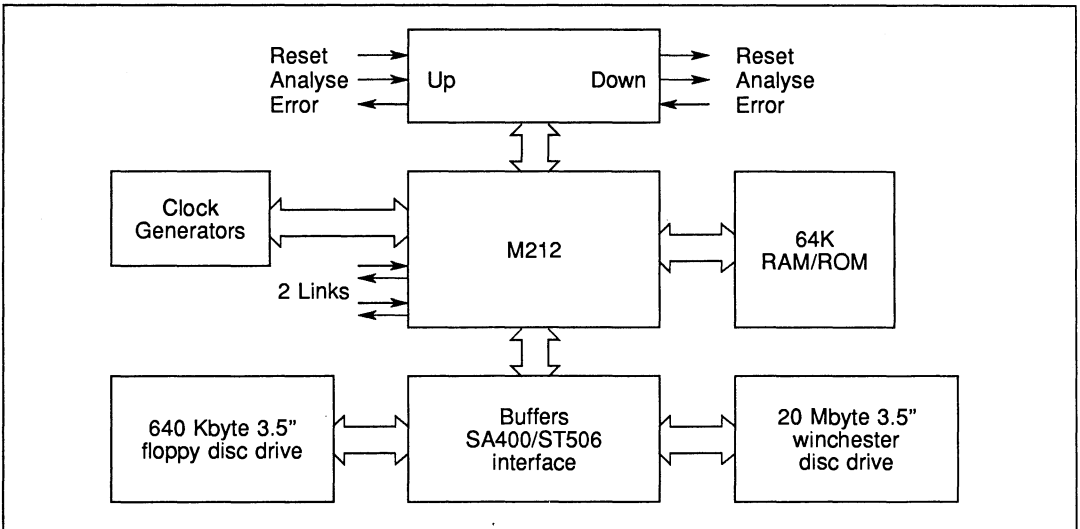
- IMS M212 disk controller transputer, with two standard links and 64 Kbyte of static RAM
- SA400/ST506 standard disk drive interface with buffering
- 20 Mbyte 3.5 inch Winchester disk drive
- 1 Mbyte 3.5 inch floppy disk drive
- Double Extended Eurocard

GENERAL DESCRIPTION

The IMS B005 board allows the user to evaluate and demonstrate the use of the M212 disk controller transputer.

The IMS M212 is able to control up to four disk drives via the industry standard SA400/ST506 interfaces. Two drives are present on the IMS B005 and provision has been made for connecting other drives if the user so desires, or to change either of the drives on the board (for instance use two Winchester drives).

The external memory interface can address 64K of memory space; as supplied this memory is static RAM (two 32K x 8 devices), but it is possible to replace one or both with EPROM if required. The external memory interface may be switch programmed for different speeds.

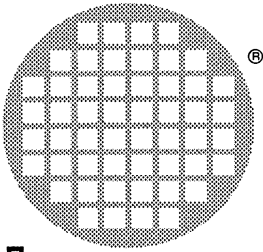


4.1 IMS B005 Double Extended Eurocard product overview

4.1.1 Ordering Information

Description	Order Number
IMS B005 Double Extended Eurocard	IMS B005-1

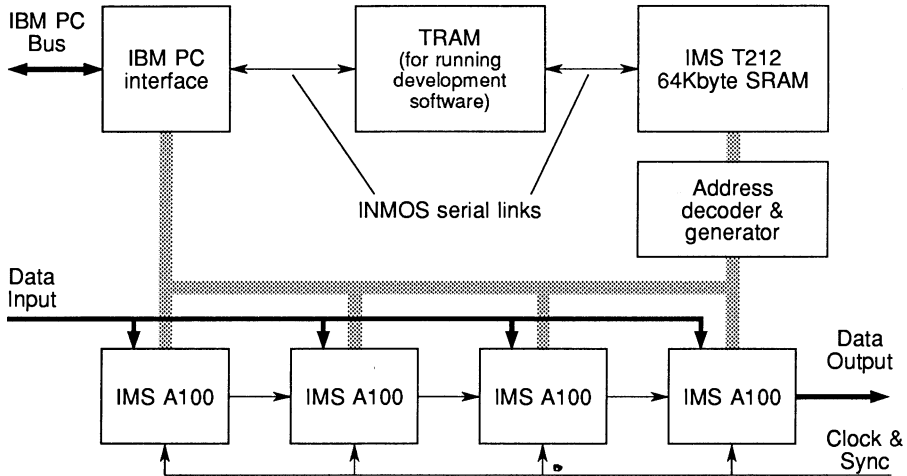
Table 4.1 Ordering information



inmos

IMS B009

Signal processing board



Features

- High performance Digital Signal Processing development board for both real-time and non-real time compute-intensive applications
- Cascade of four IMS A100s
- Up to 1280 Million Operations Per Second (MOPS) capability
- Up to 10MSamples/sec continuous data throughput
- Fully programmable using an IMS T212 16 bit transputer with 64Kbyte SRAM
- Option to install TRAM
- Transputers arrayable for high performance pipelined systems
- General purpose address mapper (Look Up Table) for data sequencing
- Data supplied from internal (i.e. software/file) or external sources
- Controllable from IBM PC applications under MS-DOS or other transputer systems
- IMS A100 cascade accessible directly from IBM PC bus
- Complete DSP development environment available, including IMS A100 and IMS B009 software simulators
- Compatible with full transputer board family

4.2 IMS B009 DSP System Evaluation Board product overview

4.2.1 The IMS B009 Evaluation Board

The IMS B009 can be used to evaluate and implement a wide range of high performance DSP techniques. It can also be used by OEMs as a component for building high performance, flexible, DSP systems, where the production quantities do not justify development of a specific DSP board.

The IMS B009 is an IBM PC (XT or AT) add-in board containing 4 IMS A100 signal processors controlled by an IMS T2. The 4 IMS A100s can be used to implement 128 tap FIR filters, convolvers or correlators, on 16 bit data, with 16 bit coefficients at rates up to 2.5 M samples/second, or at up to 10 10M samples/second with 4 bit coefficients.

The IMS A100s can be controlled and configured either directly from the IBM PC or, for much greater performance, by the IMS T2. Data flow through the IMS A100s can be controlled by the IBM PC, the IMS T2xx or directly from an external digital signal source, via a DIN 41612 edge connector. This last option allows the IMS A100s to process data at rates up to 10MHz.

The IMS T2xx connects to 64Kbytes of fast SRAM. The interface between the IMS T2, the SRAM, and the IMS A100s is designed to allow the IMS T2xx to move data through the IMS A100s at speeds up to 1.25 M samples/second. An address mapping table allows the IMS T2xx to perform complex data sequencing tasks at high speeds. Each of the 4 transputer links on the IMS T2xx can be used to transfer data between the IMS B009 and other transputer systems at up to 0.8 Mbytes/second simultaneously using more than one link for data I/O can provide data transfer rates of several Mbytes/second.

The IMS B009 is a TRAnsputer Module (TRAM) motherboard. A single TRAM, up to size 4, can be installed. For example, an IMS B404 TRAM can be used to provide 2 Mbytes of data storage and additional (possibly floating point) data processing. This same TRAM could be used to run software packages such as the IMS D700 Transputer Development System and the IMS D703 DSP Development System. The IMS B009 (with a suitable TRAM) thus provides the basis for both a Transputer and a DSP development workstation.

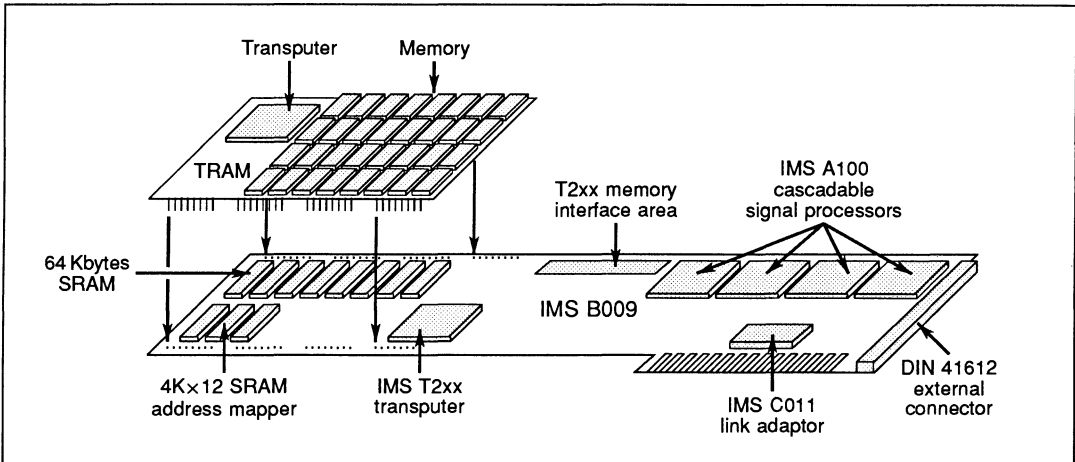


Figure 4.1 IMS B009 key components

4.2.2 Board Description

The IMS B009-1 contains a cascade of four IMS A100s, an IMS T2xx 16 bit transputer with 64Kbyte SRAM, and a socket for a standard TRAM.

An INMOS TRAM (e.g. IMS B404) provides a general purpose host processor, capable of supporting the full INMOS Occam 2 Transputer Development System, and the IMS D703 DSP Development System. The IMS T2is used as a high performance controller for the IMS A100 cascade. Figure 2 shows the board with the optional TRAM, and the configuration of the IMS A100 cascade.

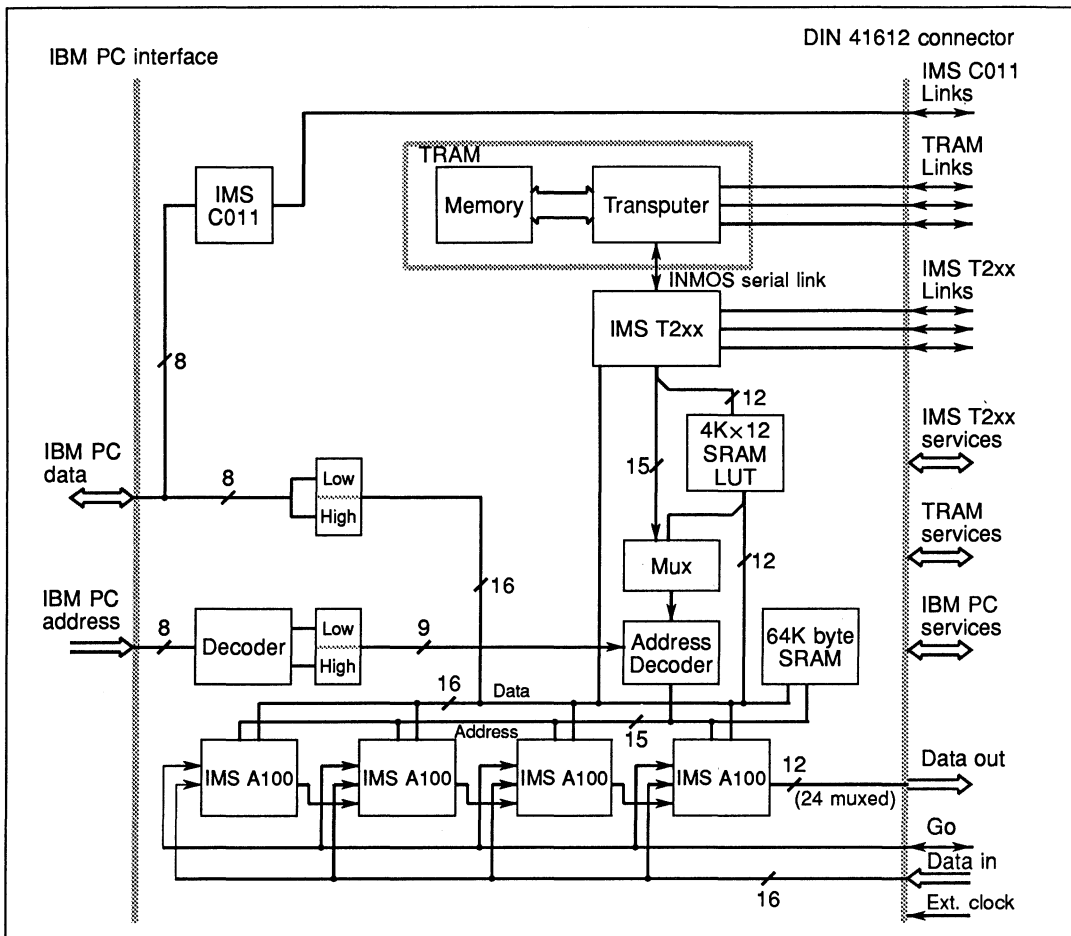


Figure 4.2 IMS B009 Detailed Block Diagram

A 4Kx12 'address mapper' is provided for high speed generation of arbitrary address sequences. This mapper can be applied at any time to any addresses generated in the positive address space of the IMS T2, without any performance degradation. Thus, arbitrary data sequences can be preloaded, and applied at the appropriate point during data processing.

The IMS T2xx can be connected to any other transputer (including the optional IMS T414 on the IMS B009-2) with one or more standard INMOS serial links, each link being capable of approx. 0.9MBytes/sec in each direction, and operation in full duplex. The transputer links can also be used to connect to other transputer evaluation boards, or for arraying IMS B009s to form a high bandwidth signal processing pipeline.

The TRAM/B009-1 combination offers a powerful concurrent processing environment, with preprocessing operations such as data pre/post ordering handled by the TRAM, whilst the IMS T2xx drives the IMS A100s. For highest performance, external ports are provided, enabling users to supply real time data to the A100 cascade, and output processed data at full speed. Thus, real time processing can be implemented with a minimum of additional hardware.

In order to maximise the range of applications of the IMS B009, most of the key control and data signals are brought to either the 96 way DIN 41612 connector, or to an internal connection area. This enables users to construct DMA interfaces to all devices on the IMS T2xx memory interface bus. Thus, a wide range of real time interfaces can be realised, making the IMS B009 ideal for general laboratory use or for prototyping final systems.

Input data can thus be supplied from one of four sources:

- External data port (10MSamples/sec continuous)
- IMS T2xx memory interface (approx. 5MBytes/sec burst)
- Transputer link (approx. 0.8MBytes/sec x 4 burst)
- IBM PC bus (approx. 0.2MBytes/sec burst)

Due to the relatively small power supplies provided with some IBM PC compatibles, special links have been provided to isolate the V_{cc} plane from the IBM PC power pins, and to provide external power directly via the DIN 41612 connector. This enables several IMS B009s to be used in a standard IBM PC chassis without danger of exceeding either power supply or backplane ratings.

4.2.3 Programming

The IMS B009 enables users to exploit the flexibility of the IMS A100 under a standard OCCAM 2 environment, by running the IBM PC Transputer Development System (IMS D700) on the TRAM located on the board itself. In this way, high performance DSP systems can be realised, using high level languages throughout. The IMS D703 DSP Development System, supplied in both source code and binary form, demonstrates how to make best use of the various addressing modes and facilities of the board.

The board can also be treated as a peripheral to the IBM PC, responding to commands sent to it on the PC bus. This mode of operation disables the transputers and limits data rates to those attainable on the standard IBM PC bus, but does enable users to evaluate the potential of attaching IMS A100s directly to an IBM PC, controlled by a normal PC based program.

Alternatively, using the IMS B009 driver program supplied with the IMS D703, the IBM PC application can boot the IMS T2xx directly. It can then use the driver program in exactly the same way as any transputer application, communicating with the IMS T2xx via the link adaptor. This approach provides far higher performance for IBM PC hosted applications than controlling the IMS A100s directly via the PC bus.

4.2.4 Product summary

The IMS B009-1 comprises four IMS A100s, one IMS T2xx 16-bit transputer with 64Kbyte SRAM, and the 4Kx12 address mapper. It also contains an unpopulated socket for a TRAM (up to size 4).

A comprehensive suite of documentation is supplied with each system, including full descriptions of the board design, software users and reference manuals, and a set of application notes. Test software is also provided, which performs extensive diagnostics of all functional components of the board.

4.2.5 Technical summary

Board ready for installation in a single IBM PC XT or AT system unit expansion slot

Four IMS A100-G21S cascadable signal processors

One IMS T2xx 16 bit transputer with 64 kbytes SRAM

10 or 20 MBit/sec INMOS link transmission speeds

DIN 41612 96 pin I/O connector

A100 signals:

Data in/out
Clock/Go/OutRdy

Transputer signals:

TRAM, T2xx Reset/Analyse/Error
1 INMOS link from IMS C011
3 INMOS links from IMS T2xx
3 INMOS links from TRAM

+5V, ground

Cables (suitable for connection to all INMOS evaluation boards)

INMOS links
Up/Down Reset/Analyse/Error cables
Standard Jumpers

Power supply required (from IBM PC or externally)

+5V (approx. 4 amps with TRAM)
Ground

Note: *the IMS B009-1 can operate with external power supplies if required.*

4.2.6 Ordering details

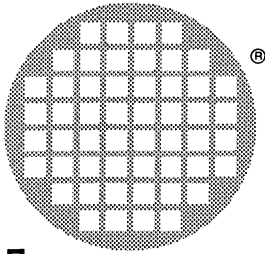
Product	Part number
B009 Evaluation board	IMS B009-1



Development systems



Software Development Tools



inmos

IMS D705

IMS D605

IMS D505

occam 2

Toolset

Software for IBM PC and NEC PC

DESCRIPTION

The **occam 2** toolset is a complete cross development system for building and debugging occam and mixed language programs for transputers.

KEY FEATURES

- Complete **occam 2** development system
- Targets to mixed networks of IMS T212, T222, M212, T414, T425, T800
- Support for machine code inserts
- Support for mixed language developments
- Tools to aid system building and ensure program consistency
- Source level debugging tools
- Support for program loading and remote debugging
- Support for EPROM programming
- Support for teams of developers
- Easily used with project management and source control utilities
- Source and binary files directly compatible across PC, VAX and SUN hosts

APPLICATIONS

- Single and multi-processor embedded systems
- Scientific programming
- Framework for mixed language developments
- Framework for running existing applications (written in C, FORTRAN or Pascal) on transputer networks and accelerator boards
- Evaluation of concurrent programming and transputers

5.1 OCCAM 2 toolset product overview

5.1.1 Product overview

The OCCAM 2 toolsets provide complete OCCAM 2 cross-development systems for transputer targets. They can be used to build parallel programs for single transputers and for multi-transputer networks consisting of arbitrary mixtures of transputer types. Programs developed with the toolset are both source and binary compatible across the host development machines.

The occam 2 toolset is available for three development platforms.

D705 IBM PC occam 2 toolset

D605 VAX occam 2 toolset

D505 SUN 3 occam 2 toolset

occam 2 development system

The OCCAM 2 compiler supports the full OCCAM 2 language as defined in the OCCAM 2 reference manual. In addition to type checking the compiler will check programs to ensure that variables and communication channels are being used correctly in parallel components of a program. This detects many simple programming errors at compile time. The language provides support for low-level programming, including the allocation of variables to specific memory addresses, access to timers and the inclusion of transputer machine code.

The compiler will generate code for the full range of transputer types; IMS T212, T222, M212, T414, T425, and T800. It can exploit the cases when the code for sets of these processors is identical. This reduces the volume of code required in libraries and allows you to build programs which you can guarantee will run on a range of processors.

A program may be compiled in one of several 'error modes' which determine the behaviour of the program when a run-time error occurs. A mode which supports the use of the debugger may be chosen; in this mode the network will come to a halt when a run-time error occurs. Another mode allows all error checking to be removed from time critical or proven components and to call these components from within a checked system.

The processor target, error mode and other compilation options are specified by command line switches.

Collections of procedures and functions can be compiled separately with the OCCAM 2 compiler. Separately compiled units may be collected together into libraries. The linker is used to combine separately compiled units into a program to run on a single processor. The linker supports selective loading of library units, and the linking of C, FORTRAN and Pascal components. The logic of OCCAM program fragments may be checked using the transputer simulator, prior to building the complete system.

To build a multi-transputer program you must describe the distribution of procedures over processors. In OCCAM this distribution is explicit in the so-called 'configuration' description. This description is processed by a tool called the configurer. The configurer creates all the necessary bootstrap and routing information to load the entire network, and stores this, along with the compiled code, in a program file.

The server is used to load programs on to transputer networks. Once loaded, the programs start automatically. The server supports access to the host terminal and file system from the transputer network.

If the transputer network halts because of a run-time error, or if you bring the server to a halt by entering a control break, you may use the symbolic network debugger to investigate the state of the network in terms of program source texts.

The compiler and linker provide features which allow optimal use of the transputer's on-chip RAM. The mathematical function libraries are precisely documented, fast and accurate; all mathematical functions recommended by the IEEE are supported.

Support for mixed language developments

It is often appropriate in the development of a large system to use a mixture of programming languages. You may either already have software which you want to reuse, or find that an algorithm may be more easily expressed in one particular language. The OCCAM programming model provides an excellent vehicle for building mixed language systems where the components built in each language can be clearly defined with a simple interface. The OCCAM 2 toolset can be used to bind these components together and distribute them over a network of transputers.

The OCCAM 2 toolset allows you to build parallel systems using C, FORTRAN and Pascal whilst still using the standard language definitions. A process can be built using the C, FORTRAN and Pascal compilers in a form equivalent to an OCCAM process (each language is supplied with a library which provides channel input and output functions). The result of linking a C, FORTRAN or Pascal program is identical to the result of compiling the equivalent OCCAM process with the OCCAM 2 compiler, and can therefore be called from OCCAM in exactly the same way as an OCCAM process. In particular, using the configuration language you can define the inter-connections between these processes and distribute them over a network of transputers.

When debugging a mixed language program the symbolic network debugger can be used to analyse a halted network. It will automatically locate the source line (for each processor) where the network halted. It can do this in any of the languages, and will allow you to backtrace through the sequence of function or procedure calls executed prior to halting.

C, FORTRAN and Pascal compilers are available separately from INMOS.

System building and program consistency

The OCCAM 2 language has been defined so that extensive security checks can be performed at compile time providing a degree of reliability not found in other languages. In addition to full type checking the language defines a comprehensive set of rules which ensure that variables and communication channels are correctly used in parallel components of the program.

The OCCAM 2 compiler implements the full OCCAM 2 language as defined in the OCCAM 2 reference manual. All checks are performed both within a separately compiled unit, and across separately compiled units. This detects many programming errors at compile time and avoids common pitfalls in other separate compilation systems where external references are not checked against their definitions.

The toolset provides support for the use of the make program. The make program was originally a UNIX tool but is now widely available for the PC both as a public domain program and from well respected commercial sources. The make program operates from a 'makefile' which defines the rules for building programs, and the dependencies between the program components. When make is executed it will determine which components need rebuilding by identifying components which have dependents with a more recent time and date, it then applies the build rules to create updated versions of these components. This means that the programmer can make changes to a number of source files and then by just typing 'make' bring his program back to a consistent state; the compiler, librarian, linker and configurer will all be invoked automatically as necessary.

The toolset provides 'imakef' a tool which will automatically generate make files for OCCAM programs and libraries. You can give imakef the name of a multi-transputer executable file and it will create a makefile for the complete program by using suffix conventions and parsing source files for compiler directives. Then by simply typing 'make' the system will be rebuilt.

The linker and librarian can both be driven by command files. The 'imakef' tool will also automatically generate linker command files. This means you do not have to work out the list of binaries and libraries required to build a program, nor should you discover undefined references at link time.

Source level debugging tools

The toolset includes two debugging tools: the symbolic network debugger, and the transputer simulator.

The symbolic network debugger can be used to examine the state of a transputer in a network, in terms of the source of the program that was running on it. After a program has halted or been interrupted, the memory

state can be preserved so that the debugger can be run. The debugger will allow you to analyse the network directly, or to save the complete state of the network for future analysis.

The debugger can be used to determine the position of the halted process, and of any other process waiting on queues, in any processor in the network. These positions are displayed as lines in the OCCAM C, FORTRAN or Pascal source. The values of OCCAM variables can be displayed. For each process it is possible to trace back through the sequence of procedure calls which led to the halted position.

You can also; inspect process and timer queues, inspect register and memory location values, browse memory in different forms including disassembled machine code.

The debugger works with exactly the same code as will run in your final product; there is no additional code inserted to support debugging. This avoids the cases where the program works when debugging is included, but fails when the debugging is removed.

The debugger can be used for programs that run on your PC transputer board or for programs which run on a transputer network attached to your transputer board by a link.

The transputer simulator simulates a single T414 processor connected to the host file server. It provides single-step, breakpoint and watchpoint facilities for OCCAM programs, and low level features to access to memory addresses, process queues and internal registers. The simulator is used for testing the logical correctness of OCCAM program fragments prior to building the complete system. As the logical correctness of an OCCAM program is not dependent on the type of transputer on which it will run, the simulator can be used to test programs targetted at any type of transputer.

Support for teams of developers

The OCCAM 2 toolset has been designed to support teams of developers.

There are consistent versions of the OCCAM 2 toolset on the PC, VAX and the SUN. The tools operate in the same way for each host. The same sources and binaries can be moved between machines and used without change. Mechanisms which avoid the need for directory paths to be included in compiler directives have been implemented in each toolset. This allows you to move from PC to VAX or SUN systems as the project grows, or to develop systems on VAX or SUN and still support debugging using a PC.

The tools allow you to take advantage of multi-user facilities of PC networks and VAX and SUN operating systems. The tools are easily integrated with other development tools such as make, source control systems, and text processing tools.

The library system provides a suitable basis for sharing code between development teams. It provides a coherent scheme for collecting separately compiled units within a single library file. It will allow the same procedure, compiled for different processor types and error modes, to be included in the same library.

A binary lister program is provided to generate consistent documentation of library interfaces and cross reference tables. You can also use the binary lister to display library interfaces without reference to the source texts. The compiler supports a comment directive which allows comments to be included in binary files and subsequently displayed with the binary lister. Typically these comments contain a version number, date of last update and a short description of the separately compiled unit.

5.1.2 OCCAM 2 toolset product description

The OCCAM 2 toolset product consists of the following components:

Documentation

The OCCAM 2 Toolset is supported with comprehensive user documentation.

Delivery manual

The delivery manual provides instructions for installing the software on your machine, and checking that the installation has been successful. The delivery manual will define the implementation limits for the products.

User manual

The user guide and reference manual are provided in a single volume. The user guide provides a complete introduction to the software and is designed to be read while starting to use the software. A number of examples are described to get you started. The reference manual describes the behaviour of the tools and defines the user interfaces and software libraries in detail; it is supplemented by tabulations of useful information in appendices, with a glossary and bibliography. The manual is fully indexed.

occam 2 toolset handbook

This quick reference guide is for the experienced developer. It provides a summary of all the toolset components in terms of the command line syntax and definition of options. It also provides a summary of procedures and functions in the commonly used libraries.

occam 2 reference manual

This book, published by Prentice Hall (ISBN 0-13-629312-3), is the definitive reference manual for the OCCAM 2 language.

Tutorial introduction to OCCAM

This book, published by BSP Professional Books (ISBN 0-632-01847-X), contains a tutorial introduction to concurrent programming using the OCCAM language and also a concise formal definition of the syntax and semantics of the language.

Software tools

This section gives an outline of the function of each of the software tools in the toolset.

icheck

The checker provides a fast tool for developing syntactically correct OCCAM program units. It carries out all checks made by the compiler, including checks for externally called procedures and functions. This tool runs significantly faster than the compiler. It has an effective error recovery algorithm which provides multiple error messages in a useful manner.

occam

The OCCAM 2 cross-compiler supports the OCCAM 2 language as defined in the OCCAM 2 language reference manual. It can generate code for IMS T212, T222, M212, T414, T425, and T800 processors. It supports separate compilation and performs full security checks across separate compilation boundaries. The compiler supports machine code inserts.

ilibr

The librarian allows separately compiled units to be grouped together in a single file. Libraries can contain definitions of the same procedure or function compiled for different processor targets and in different error modes. Libraries provide the basis for the selective loading mechanisms of the linker.

ilink

The linker composes separately compiled units, resolving external references, to give a single code unit. This is typically used to build the code for a single processor.

ilist

The binary lister is used to generate documentation of object code and libraries. It can produce listings of: the syntactic interface to procedures and functions held in the library; table of units contained in a library including the processor type and error modes for which they were compiled; external reference table.

iboot

The iboot tool prepends bootstrap loading code to the executable code for a single processor program.

iconf

The configurer builds multi-transputer programs from a configuration description and units built by the linker. The configurer adds all loading and routing code so that a complete network of transputers can be loaded through the server.

imakef

The imakef tool performs source dependency analysis of OCCAM programs and OCCAM libraries. The tool can automatically generate makefiles and linker command files. It will also check for circularities in the references between separately compiled units. This program is also provided in C source to allow modifications to support tools similar to the make program.

isim

The T414 simulator is a tool for debugging the logic of OCCAM program fragments. It supports breakpoint and variable watchpoint facilities, and allows inspection of the simulated process queues, registers and memory.

idebug

The symbolic network debugger allows a halted network to be analysed in terms of the program source. No additional code is generated by the compiler to support the operation of the debugger. The debugger supports mixed language programs. The debugger will operate on the target transputer network or on the transputer plug-in board. The debugger can save the state of a network for future analysis.

idump

When debugging a network which includes the first transputer on the plug-in board, the idump tool is used to save the contents of memory from the first transputer in the network, so that the debugger can be loaded and executed on the first transputer. The debugger then references the dump file to answer questions about the first transputer.

iskip

The iskip tool is used to support running programs on transputer networks completely separate from your development system. It supports loading programs to a separate network and carrying communications between the program and the server when it is running.

iserver

The server provides two basic functions. Firstly it will load programs to a transputer network, and secondly it will support access to the host file and terminal system. The server is also supplied as C source code to allow user extensions and porting to different hosts.

Software libraries**occam.lib**

This is the basic OCCAM run-time library. It includes: multiple length arithmetic functions; floating point functions; IEEE arithmetic functions; 2D block moves; bit manipulation and CRC; code execution; arithmetic

instructions. The compiler will automatically reference these functions if they are required.

maths.lib

Single and double length mathematics functions (including trigonometric functions). These libraries use floating point arithmetic and will produce identical results on all processors.

t4maths.lib

Mathematical functions optimised for the T414 and T425 processor. These functions provide slightly different results to the maths library above but within the accuracy limits of the function specifications.

string.lib

String conversion procedures. The OCCam source code is also provided.

hostio.lib

Procedures to support access to the host terminal and file system through the file server. The OCCam source code is also provided.

msdos.lib

Procedures to access certain DOS specific functions through the file server. The OCCam source code is also provided.

process.lib

Processes to support the use of the file server. Includes: server multiplexor to allow parallel processes to access the file server; support processes for interfacing C, FORTRAN and PASCAL to the file server; processes for interfacing to IMS B001, B002, and B006 transputer evaluation boards including UART and serial line drivers. The OCCam source code is also provided.

xlink.lib

These procedures allow access to external links. They include input and output routines which return error if the link is not connected, or a communication is not completed.

Programming examples

The examples given in the user guide are provided in machine readable form. They provide a suitable basis for getting started with the toolset.

5.1.3 D700D transputer development system support

The D700D is a fully integrated PC development system for OCCam and transputers. A number of tools are provided with the OCCam 2 toolset to support its use in conjunction with the D700D.

- D700D folded file flattener
- Utility to import toolset compilation units as D700D SC foldsets
- Libraries to support transfer of programs from D700D to toolset

5.1.4 OCCAM 2 toolset product components summary

Documentation

- Delivery manual
- Toolset user manual
- Toolset quick reference guide
- OCCAM 2 reference manual
- Tutorial introduction to OCCAM

Software tools

- OCCAM 2 checker
- OCCAM 2 cross-compiler
- Librarian
- Linker
- Binary lister
- Add bootstrap program
- Configurer
- Makefile generator
- T414 simulator
- Symbolic network debugger
- Memory dump program
- Skip loader
- File server and loader

Software libraries

- OCCAM: standard compiler library
- maths: mathematics functions (includes sin, cos etc)
- t4maths: mathematics functions optimised to run on T414 and T425
- string: string conversion procedures
- hostio: file and terminal io procedures
- msdos: access to certain msdos calls
- process: standard OCCAM processes
- xlink: external link primitives

5.1.5 D705 IBM PC version

Although the PC tools are invoked as if they were ordinary PC resident tools they actually run on the transputer board plugged into the PC.

Operating requirements

- IBM PC XT or AT, or NEC PC-9801 with 256 Kbyte memory
- IMS B008 or IMS B015 plus IMS B404 or equivalent
- At least 1 Mbyte of transputer memory
- DOS 3.0 or later
- 4 Mbytes of free disk space

Distribution media

Software is distributed on 360 Kbyte (48TPI) IBM format floppy disks.

5.1.6 D605 VAX VMS version

Both VAX hosted and transputer hosted development tools are provided in the VAX toolset. With the exception of the symbolic network debugger which requires a target system to execute all tools are provided in both forms.

Operating requirements

For hosted cross-development and debugging you will need

- VAX VMS 4.7 or later
- 10 Mbytes of free disk space

The hosted tools are provided in object code form so that if necessary they can be re-linked when operating system upgrades demand this.

For loading target systems and remote debugging you need one of the following

- A third party interface board
- An IBM PC with DECNET connection and B008 motherboard.

A system for download and remote debugging over EtherNet is under development.

Distribution media

Software is distributed on a TK50 tape cartridge in VMS backup format.

5.1.7 D505 SUN 3 version

Both SUN 3 hosted and transputer hosted development tools are provided in the SUN 3 toolset. With the exception of the symbolic network debugger which requires a target system to execute all tools are provided in both forms. The transputer hosted tools can be run on a transputer board in a SUN 4 or SUN 386i machine.

Operating requirements

For hosted cross-development and debugging you will need

- Sun 3 Workstation
- SunOS 4.0 or later
- 10 Mbytes of free disk space

For loading target systems and remote debugging you need one of the following

- IMS B011 VME Card
- IMS B014 VME Motherboard with 2 Mbyte IMS B404 TRAM

A system for download and remote debugging over EtherNet is under development.

Distribution media

Sun Data Cartridge (Quarter inch tape) in tar format.

5.1.8 Associated products

D711 3L C compiler (PC)

D712 3L PASCAL compiler (PC)

D713 3L FORTRAN compiler (PC)

D611 3L C compiler (VAX)

D613 3L FORTRAN compiler (VAX)

D511 3L C compiler (SUN 3)

D513 3L FORTRAN compiler (SUN 3)

For more details of these products refer to their respective data sheets.

5.1.9 Licencing information

No licence fee is charged for including the binary of the INMOS libraries in software products. Example programs, and other sources provided, may be included in software products. Full licensing details are available from INMOS.

The following products are also available for licensing:

D705-B D705 Binary Distribution Licence

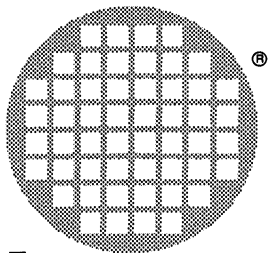
D705-S D705 Source Release

5.1.10 Error reporting and field support

Software problem reports are included with the software.

INMOS has a world-wide network of sales offices, providing support for INMOS products. In some areas the support functions may be taken over by distributors or other third parties.

Customers requiring a formal support contract should contact their local INMOS sales office.



inmos

IMS D711

IMS D611

IMS D511

Parallel C

DESCRIPTION

The Parallel C compiler allows you to program single transputers and networks of transputers in C.

KEY FEATURES

- Kernigan and Ritchie C
- Compatible with INMOS occam 2 Toolsets (IMS D505, D605, D705)
- Produces code for IMS T414, T425, T800, T801, T805 transputers
- Transputer channel I/O with timeouts
- Concurrent Tasks
- Semaphores
- Ability to allocate stack in on-chip RAM

5.2 Parallel C compiler product overview

5.2.1 Product overview

The 3L Parallel C compiler supports Kernigan and Ritchie C. Consequently existing C applications can be easily ported to transputer systems. The Parallel C compiler can be used by itself to develop single and multi-transputer systems. Alternatively, it can be used in conjunction with the INMOS occam Toolsets for mixed language developments over single and multiple transputers.

Support for parallelism

Parallel C supports two types of components which can be used to build parallel programs: tasks and threads.

Tasks communicate by message passing over channels. Configuration tools are provided to distribute tasks over transputer networks. A task may not be distributed over more than one processor.

Tasks can contain multiple parallel execution threads. Threads are light weight processes which can be created at run time. Threads within the same task can communicate using channels or shared memory with accesses synchronised using semaphores.

There are two configurers provided in the Parallel C package. A "flood-filling" configurer supports "processor farm" applications. This configuration method will allow the same software to run unchanged on any network. An alternative configuration method involves the specification of the network in a configuration file by the user.

Using C with the occam 2 toolset

The Parallel C compiler can be used by itself to develop single and multiple transputer systems without the need to write any occam code. However there are several advantages to using the occam Toolset with the C compiler.

The occam Toolset products available for SUN, VAX and PC complement the Parallel C compiler by offering the following additional facilities.

- Several communicating C processes can be run in a single transputer without using threads. This encourages users to write software which could later be re-configured to run on more processors.
- Toolset configuration gives the user the added security of the occam configuration language.
- More control is available over the amount of workspace available to each C process, and where the workspace is located.
- C processes can use and share compiled occam code and libraries on the same processor
- The occam Toolset provides a post-mortem symbolic debugger which can be used to assist debugging in an arbitrarily complex network. Should run-time errors occur within a C process, the debugger can symbolically locate to the source line that caused the error flag to be set after the network has halted. It is then possible to backtrace out through the source or object code to locate the cause of the error (although the contents of C variables are not available with this tool).

5.2.2 3L Parallel C description

The 3L C consists of the following components:

Documentation

The 3L C is supported with comprehensive user documentation.

Delivery manual

The delivery manual provides instructions for installing the software on your machine, and checking that the installation has been successful. The delivery manual will define the implementation limits for the products.

User manual

The user guide and reference manual are provided in a single volume. The user guide provides a complete introduction to the software and is designed to be read while starting to use the software. A number of examples are described to get you started. The reference manual describes the behaviour of the tools and defines the user interfaces and software libraries in detail; it is supplemented by tabulations of useful information in appendices, with a glossary and bibliography. The manual is fully indexed.

Software tools

This section gives an outline of the function of each of the software tools in the toolset.

C compiler

The compiler implements the full Kernigan and Ritchie C language with several extensions.

- Multi-threaded synchronised tasks (synchronised by semaphores or transputer channels)
- Access to transputer timer functions and channel I/O
- Microsoft C compatible DOS access functions (for PC version)
- Ability to include in-line transputer assembly language
- 31 character identifiers which can contain a \$

ilibr

The librarian allows separately compiled units to be grouped together in a single file. Libraries can contain definitions of the same procedure or function compiled for different processor targets and in different error modes. Libraries provide the basis for the selective loading mechanisms of the linker.

ilink

The linker composes separately compiled units, resolving external references, to give a single code unit. This is typically used to build the code for a single processor.

decoder

The Decoder Utility allows source-level disassembly of object files generated by the compiler.

iboot

The iboot tool prepends bootstrap loading code to the executable code for a single processor program.

config

The configurer allows multiple C tasks to be executed in parallel. The user supplies a configuration file which describes the target transputer network in terms of processors, links and connections. This file should also identify which software is to be executed on which processor. The placement of C tasks on the processors can be easily altered by changing the configuration file. No recompilation or relinking is necessary. This allows simple development on a single transputer and later execution on the network by altering the configuration file.

fconfig

This configurer provides the flexibility of developing software that will run on any network of transputers with sufficient memory. The configuration of the network need not be decided when the program is written. Instead, this configurer can "flood-fill" an arbitrary network with copies of a user-supplied "worker" program. A "master" task is used to generate work packets which are communicated to the "worker" programs on each node. These "worker" programs will route the work to any free processor on the network and return the results to the "master".

iserver

The server provides two basic functions. Firstly it will load programs to a transputer network, and secondly it will support access to the host file and terminal system. The server is also supplied as C source code to allow user extensions and porting to different hosts.

Software libraries

occam.lib

This is the basic OCCAM run-time library. It includes: multiple length arithmetic functions; floating point functions; IEEE arithmetic functions; 2D block moves; bit manipulation and CRC; code execution; arithmetic instructions. The compiler will automatically reference these functions if they are required.

5.2.3 3L C components summary

Documentation

- Delivery manual
- User manual

Software tools

- C compiler
- Librarian
- Linker
- Decoder
- Add bootstrap program
- Configurer
- Flood-fill configurer
- File server and loader

Software libraries

- Full C run-time library
- Reduced C run-time library

5.2.4 D711 IBM PC version

Although the tools are invoked as if they were ordinary host resident tools they actually run on the transputer board connected to the host.

Operating requirements

- IBM PC XT or AT, or NEC PC-9801 with 512 Kbyte memory
- IMS B008 or IMS B015 plus IMS B404 or equivalent
- At least 1 Mbyte of transputer memory
- DOS 3.2 or later
- 2 Mbytes of free disk space

Distribution media

360 Kbyte (48TPI) IBM format floppy disks.

5.2.5 D611 VAX VMS version

Although the tools are invoked as if they were ordinary host resident tools they actually run on the transputer board connected to the host.

Operating requirements

- VAX VMS 4.7 or later
- 2 Mbytes of free disk space
- A third party interface board
- INMOS transputer motherboard plus IMS B404 or equivalent.

A system for download over EtherNet is under development.

Distribution media

TK50 tape cartridge in VMS backup format.

5.2.6 D511 SUN 3 version

Although the tools are invoked as if they were ordinary host resident tools they actually run on the transputer board connected to the host.

Operating requirements

- Sun 3 Workstation
- SunOS 4.0 or later
- 2 Mbytes of free disk space
- IMS B011 VME Card or
- IMS B014 VME Motherboard with 2 Mbyte IMS B404 TRAM

A system for download and remote debugging over EtherNet is under development.

Distribution media

Sun Data Cartridge (Quarter inch tape) in tar format.

5.2.7 Associated products

D705 occam 2 toolset (PC)

D605 occam 2 toolset (VAX)

D505 occam 2 toolset (SUN)

D713 3L FORTRAN compiler (PC)

D613 3L FORTRAN compiler (VAX)

D513 3L FORTRAN compiler (SUN 3)

D712 3L PASCAL compiler (PC)

For more details of these products refer to their respective data sheets.

5.2.8 Licencing information

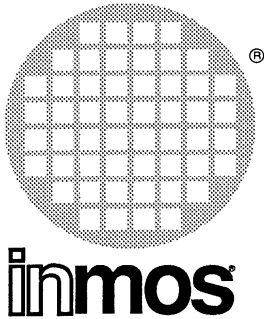
No licence fee is charged for including the binary of the INMOS libraries in software products. Example programs, and other sources provided, may be included in software products. Full licensing details are available from INMOS.

5.2.9 Error reporting and field support

Software problem reports are included with the software.

INMOS has a world-wide network of sales offices, providing support for INMOS products. In some areas the support functions may be taken over by distributors or other third parties.

Customers requiring a formal support contract should contact their local INMOS sales office.



IMS D713 IMS D613 IMS D513 Parallel FORTRAN

DESCRIPTION

The Parallel FORTRAN compiler allows you to build programs for single transputers and networks of transputers in FORTRAN.

KEY FEATURES

- Full ANSI standard FORTRAN X3.9-1978 language support
- Compatible with INMOS occam 2 Toolsets (IMS D505, D605, D705)
- Produces code for IMS T414, T425, T800, T801, T805 transputers
- Transputer channel I/O with timeouts
- Concurrent Tasks
- Semaphores
- Ability to allocate stack in on-chip RAM

5.3 Parallel FORTRAN compiler product overview

5.3.1 Product overview

The INMOS Parallel FORTRAN compiler is a full ANSI Fortran 77 compiler. Consequently existing FORTRAN applications can be easily ported to transputer systems. The Parallel FORTRAN compiler can be used by itself to develop single and multi-transputer systems. Alternatively, it can be used in conjunction with the INMOS occam Toolsets for mixed language developments over single and multiple transputers.

Support for parallelism

Parallel FORTRAN supports two types of components which can be used to build parallel programs: tasks and threads.

Tasks communicate by message passing over channels. Configuration tools are provided to distribute tasks over transputer networks. A task may not be distributed over more than one processor.

Tasks can contain multiple parallel execution threads. Threads are light weight processes which can be created at run time. Threads within the same task can communicate using channels or shared memory with accesses synchronised using semaphores.

There are two configurers provided in the Parallel FORTRAN package. A "flood-filling" configurer supports "processor farm" applications. This configuration method will allow the same software to run unchanged on any network. An alternative configuration method involves the specification of the network in a configuration file by the user.

Using FORTRAN with the occam 2 toolset

The Parallel FORTRAN compiler can be used by itself to develop single and multiple transputer systems without the need to write any occam code. However there are several advantages to using the occam Toolset with the FORTRAN compiler.

The occam Toolset products available for SUN, VAX and PC complement the Parallel FORTRAN compiler by offering the following additional facilities.

- Several communicating FORTRAN processes can be run in a single transputer without using threads. This encourages users to write software which could later be re-configured to run on more processors.
- Toolset configuration gives the user the added security of the occam configuration language.
- More control is available over the amount of workspace available to each FORTRAN process, and where the workspace is located.
- FORTRAN processes can use and share compiled occam code and libraries on the same processor
- The occam Toolset provides a post-mortem symbolic debugger which can be used to assist debugging in an arbitrarily complex network. Should run-time errors occur within a FORTRAN process, the debugger can symbolically locate to the source line that caused the error flag to be set after the network has halted. It is then possible to backtrace out through the source or object code to locate the cause of the error (although the contents of FORTRAN variables are not available with this tool).

5.3.2 3L Parallel FORTRAN description

The 3L FORTRAN consists of the following components:

Documentation

The 3L FORTRAN is supported with comprehensive user documentation.

Delivery manual

The delivery manual provides instructions for installing the software on your machine, and checking that the installation has been successful. The delivery manual will define the implementation limits for the products.

User manual

The user guide and reference manual are provided in a single volume. The user guide provides a complete introduction to the software and is designed to be read while starting to use the software. A number of examples are described to get you started. The reference manual describes the behaviour of the tools and defines the user interfaces and software libraries in detail; it is supplemented by tabulations of useful information in appendices, with a glossary and bibliography. The manual is fully indexed.

Software tools

This section gives an outline of the function of each of the software tools in the toolset.

FORTRAN compiler

The compiler implements the full ANSI Fortran-77 language with several extensions.

- Multi-threaded synchronised tasks (synchronised by semaphores or transputer channels)
- Access to transputer timer functions and channel I/O
- Routines to provide access to MS-DOS functions
- IMPLICIT NONE, DO WHILE and INCLUDE statements
- Bit handling intrinsic functions
- Data initialisation in type statements
- Extended range of DO loops
- Lower case in program text
- 31 character identifiers which can contain \$ and '.'

ilibr

The librarian allows separately compiled units to be grouped together in a single file. Libraries can contain definitions of the same procedure or function compiled for different processor targets and in different error modes. Libraries provide the basis for the selective loading mechanisms of the linker.

ilink

The linker composes separately compiled units, resolving external references, to give a single code unit. This is typically used to build the code for a single processor.

decoder

The Decoder Utility allows source-level disassembly of object files generated by the compiler.

iboot

The iboot tool prepends bootstrap loading code to the executable code for a single processor program.

config

The configurer allows multiple FORTRAN tasks to be executed in parallel. The user supplies a configuration file which describes the target transputer network in terms of processors, links and connections. This file should also identify which software is to be executed on which processor. The placement of FORTRAN tasks on the processors can be easily altered by changing the configuration file. No recompilation or relinking is necessary. This allows simple development on a single transputer and later execution on the network by altering the configuration file.

fconfig

This configurer provides the flexibility of developing software that will run on any network of transputers with sufficient memory. The configuration of the network need not be decided when the program is written. Instead, this configurer can "flood-fill" an arbitrary network with copies of a user-supplied "worker" program. A "master" task is used to generate work packets which are communicated to the "worker" programs on each node. These "worker" programs will route the work to any free processor on the network and return the results to the "master".

iserver

The server provides two basic functions. Firstly it will load programs to a transputer network, and secondly it will support access to the host file and terminal system. The server is also supplied as C source code to allow user extensions and porting to different hosts.

Software libraries

occam.lib

This is the basic OCCAM run-time library. It includes: multiple length arithmetic functions; floating point functions; IEEE arithmetic functions; 2D block moves; bit manipulation and CRC; code execution; arithmetic instructions. The compiler will automatically reference these functions if they are required.

5.3.3 3L FORTRAN components summary

Documentation

- Delivery manual
- User manual

Software tools

- FORTRAN compiler
- Librarian
- Linker
- Decoder
- Add bootstrap program
- Configurer
- Flood-fill configurer
- File server and loader

Software libraries

- Full FORTRAN run-time library
- Reduced FORTRAN run-time library

5.3.4 D713 IBM PC version

Although the tools are invoked as if they were ordinary host resident tools they actually run on the transputer board connected to the host.

Operating requirements

- IBM PC XT or AT, or NEC PC-9801 with 512 Kbyte memory
- IMS B008 or IMS B015 plus IMS B404 or equivalent
- At least 1 Mbyte of transputer memory
- DOS 3.2 or later
- 2 Mbytes of free disk space

Distribution media

360 Kbyte (48TPI) IBM format floppy disks.

5.3.5 D613 VAX VMS version

Although the tools are invoked as if they were ordinary host resident tools they actually run on the transputer board connected to the host.

Operating requirements

- VAX VMS 4.7 or later
- 2 Mbytes of free disk space
- A third party interface board
- INMOS transputer motherboard plus IMS B404 or equivalent.

A system for download over EtherNet is under development.

Distribution media

TK50 tape cartridge in VMS backup format.

5.3.6 D513 SUN 3 version

Although the tools are invoked as if they were ordinary host resident tools they actually run on the transputer board connected to the host.

Operating requirements

- Sun 3 Workstation
- SunOS 4.0 or later
- 2 Mbytes of free disk space
- IMS B011 VME Card or
- IMS B014 VME Motherboard with 2 Mbyte IMS B404 TRAM

A system for download and remote debugging over EtherNet is under development.

Distribution media

Sun Data Cartridge (Quarter inch tape) in tar format.

5.3.7 Associated products

D705 occam 2 toolset (PC)

D605 occam 2 toolset (VAX)

D505 occam 2 toolset (SUN)

D711 3L C compiler (PC)

D611 3L C compiler (VAX)

D511 3L C compiler (SUN 3)

D712 3L PASCAL compiler (PC)

For more details of these products refer to their respective data sheets.

5.3.8 Licencing information

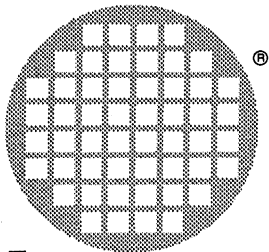
No licence fee is charged for including the binary of the INMOS libraries in software products. Example programs, and other sources provided, may be included in software products. Full licensing details are available from INMOS.

5.3.9 Error reporting and field support

Software problem reports are included with the software.

INMOS has a world-wide network of sales offices, providing support for INMOS products. In some areas the support functions may be taken over by distributors or other third parties.

Customers requiring a formal support contract should contact their local INMOS sales office.



inmos

IMS D712 Pascal

Software for IBM PC and NEC PC

DESCRIPTION

The Pascal compiler allows you to program single transputers and networks of transputers in Pascal.

Originally designed by N.Wirth for teaching purposes, Pascal has now found application in many areas including systems programming. Over the years since it was devised a great variety of Pascal dialects have evolved, many of which are mutually incompatible, and this has led in recent years to the development of international standards for the language. The Pascal compiler supplied by INMOS is one of the first to be written to these standards, namely ISO 7185 (BSI 6129:1982 Level 1).

Because the standard defines a language with limited application, the Pascal compiler provided by INMOS includes optional extensions which when invoked extend its scope in key areas. For example, modules are introduced to permit the development of large applications in a structured and partitioned manner.

Extensions included are:

- Modules for separate compilation.
- Import and export of procedures, functions and variables.
- Source file inclusion.
- The use of "\$" and "." characters in identifiers.
- Constants specified to any base between 2 and 36.
- Bit vector operators (and, or, shift, etc)
- The use of "OTHERWISE" in CASE statements.
- Non-printable characters in strings.
- Universal parameter type.
- An address function.

Ada Compilers

5.5 Ada Compilers product overview

5.5.1 Ada Compilers for the Transputer

Alslys provide validated Ada compilers for the transputer. The PC mothered compiler runs on a T8xx card. The VAX hosted compiler runs on all VAX, MicroVAX and VAXstations running under VMS. Both compilers can generate code for execution on single or linked T8xx, T4xx and T2xx transputers.

5.5.2 Features

The compilers share the following features.

- The Multi-Library Environment provides a powerful and flexible way to manage Ada development efforts and share program units even across networks.
- Detailed error messages are provided with clearly understandable diagnostic information and, optionally, possible explanations and suggested fixes.
- The High Level Optimizer excels at constraint-check removal and detection of execution-time errors at compile time.
- The Low Level Optimizer performs common sub-expression elimination and passes information to the code generator.
- Floating point instructions are generated to exploit the speed of the built-in floating point unit on the T8xx. For T4xx execution, floating point calculations are performed by software.
- Representation clauses to the bit level, address clauses (e.g. for peripheral devices), unchecked conversion and de-allocation, and interface to OCCAM 2 are among the supported low level features.
- The Binder supports unused subprogram elimination, removing all code for subprograms that are not called.
- Pragma INLINE is supported.
- Ada programs running on separate transputers can exploit rapid inter-program communication through transputer links using the CHANNEL_IO Ada interface package provided.
- Predefined Ada I/O packages are supported and performed via the INMOS iserver.
- Tasking support includes 10 levels of priority, pre-emptive scheduling, user controllable time-slicing, and a fair implementation of selective wait.
- Storage is automatically reclaimed for each access type on scope exit.
- Absolutely no execution overhead is associated with exceptions unless one is raised; call/return and block entry/exit sequences are free of exception management code and consequently very compact.

5.5.3 Recommended Configuration

Recommended Configuration for PC mothered compiler

Host:

- An IBM PC mothered T8xx with at least 4 Mbytes TRAM.
- DOS version 3.
- At least 512 Kbytes of PC memory.
- 30 Mbytes of available disk space on the PC.
- The product is supplied on both 5-1/4" high density (1.2 Mbytes) diskettes and 3-1/2" high density (1.44 Mbytes) diskettes.

Target:

- Single or multiple T2xx, T4xx or T8xx transputers.
- The INMOS iserver is required for the implementation of Ada predefined I/O.

Tools:

- IMS D705B OCCAM 2 toolset which includes the linker, run time libraries and the iserver.

Recommended Configuration for VAX hosted compiler

Host:

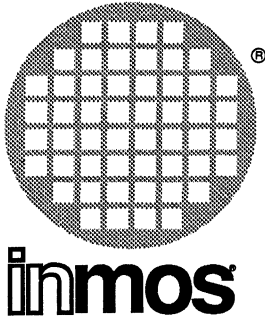
- Any VAX, MicroVAX or VAXstation running VMS version 4.7 or later.
- At least 4 Mbytes of memory.
- At least 30 Mbytes of disk space.

Target:

- Single or multiple T2xx, T4xx or T8xx transputers.
- The INMOS iserver is required for the implementation of Ada predefined I/O.

Tools:

- IMS D605A OCCAM 2 toolset which includes the linker, run time libraries and the iserver.



IMS D700 Transputer Development System

Software for IBM PC and NEC PC

DESCRIPTION

- Integrated software package for development of transputer-based systems
- Hosted on IBM PC AT or XT, NEC PC-9801 or compatible computer
- Add-in transputer board required

KEY FEATURES

- Integrated 'folding editor' user interface
- Memory-resident program development tools
- Help key, tutorial file and many introductory examples
- Full implementation of OCCAM 2 language
- Optional machine code embedded within OCCAM
- Automatic recompilation of program components
- Mathematical and input/output libraries
- Easy evolution from host development to stand alone or hosted target system
- Source-level debugging tool
- Tools for creating multi-processor programs in EPROM
- Targets to mixed networks of transputers (IMS T222, M212, T414, T425, T800 and compatible processors)

APPLICATIONS

- Development of single-processor and multi-processor embedded systems
- Development of programs for application accelerator boards
- PC-based evaluation of concurrent programming and transputers

5.6 IMS D700 Transputer Development System

5.6.1 Product overview

The IMS D700 Transputer development system (TDS) provides a full OCCAM 2 development system for transputer targets. It allows the user to build concurrent programs for single transputers, and multi-transputer programs for networks consisting of arbitrary mixtures of transputer types.

The development system software is executed on a transputer board plugged into the PC. A 'server' program running on the PC provides terminal and filing system support.

The user interface

The principal interface to the system is an editor; as soon as the system starts up the user is placed in an editing environment, and all editing, compilation and running can be carried out within that environment, by means of a set of function keys.

The editor is based on a concept called 'folding'. It provides the ability to hide blocks of text lines in a program, in the same way as a sheet of paper may be folded so that portions of the sheet are hidden. A fold contains a block of lines which may be displayed in two ways: open, in which case the lines of the fold are displayed on the screen, or closed, in which case the lines are replaced on the screen by a single fold line.

Some folds may correspond to files in the host filing system. If these files contain textual information then the fold may be opened and displayed. Folds may be nested, which provides the ability to represent a large program as a hierarchy.

A set of function keys (known as the utility function keys) can be loaded with different functions during a session. These include the OCCAM compiler, network loading software, and various file handling support functions. After the utilities have been loaded, they remain in memory and may be invoked by pressing the appropriate function keys.

OCCAM 2 compiler

The OCCAM 2 language implemented by the compiler in the Transputer development system is the full language defined in the OCCAM 2 reference manual. The compiler checks programs to ensure that variables and communication channels are being used correctly in parallel components of a program. This detects many simple programming errors at compile time. The language provides support for low-level programming of transputers, including allocation of variables to specific addresses, access to timers, and inclusion of transputer machine code.

A program may be compiled in one of three 'error modes' which determine the behaviour of the program on the occurrence of run-time errors. In particular, a mode which supports the use of the debugger may be chosen. A program may be compiled for one of a number of target processor types. IMS T222, M212, T414, T425, T800, T801, T805 and other compatible processors are all supported. It is also possible to compile code for certain classes of transputer, for example all 32 bit processors. These and other compilation parameters may be defined by completing an on-screen form called a parameter fold before running the compiler.

A program may be defined to run within the development system, using the development system to provide filing system and terminal support at run-time. Function keys are provided to load and run such a program within the development system.

A program may be defined to run on a transputer network. Such a program contains declared procedures to be loaded onto each processor in the network. If the same code is distributed to many processors, it only needs to be declared once. The distribution of procedures to processors is defined explicitly in OCCAM; this is called the 'configuration' description, and it is processed by a part of the compilation system called the configurer. From the configuration description, the configurer creates all the necessary bootstrap and routing information to load the entire network, and stores this, along with the compiled code, in a file.

Parts of a program may be compiled separately, and linked together. The unit of compilation is the 'separately compilable unit' which may contain one or more declarations of procedures or functions, accessing the calling

environment solely through their parameter lists. Collections of procedures and functions used in a variety of programs may be grouped into libraries. Libraries may be stored in a directory in the host filing system, and shared by application programs in other directories.

After components of a program have been altered, programs and libraries may be recompiled. The system will detect whether components of the program need to be recompiled, and will carry out the recompilation automatically.

Programs are compiled to make optimum use of the transputer's on-chip RAM, both when running within the development system and when running on a network.

Loading programs into transputer networks

After a network program has been compiled and configured, it may be loaded into the target network. This may be done in three ways:

- The first way is to load the network directly from the development system using the 'load network' utility, exporting the code over a link connecting the development board to the target network. The program will start as soon as it is loaded. It may communicate with the transputer running the development system by a link, and may cooperate with another program running within the development system.
- The second way is to write the program into a bootable file which can be directly loaded over a link onto the network by a server program on the host computer. The program may then optionally use the server for run-time support.
- The third way is to write the code into an EPROM which will be made part of the address space of one of the transputers in the network. Tools to write EPROMs are provided.

Debugging

The system includes a symbolic debugger which can be used to examine the state of a transputer in a network, in terms of the source of the program that was running on it. After a program has halted, or been interrupted by external action, it is possible to preserve the state of the memory so that the debugger can be run. If the program was running in the development system, the memory can be stored in a 'dump' file for future analysis. If the program was running on a target network, the state of the network may be examined from the development system.

The debugger can be used to determine the position of the halted process, and of any other processes waiting on queues, in any processor in the network. These positions are displayed as lines in the OCCAM source of the program. Values of variables may be displayed. For each process, it is possible to trace back through the sequence of procedure calls which has led to that position.

5.6.2 Product description

The product consists of the following components:

Documentation

The Transputer development system is supported with comprehensive user documentation.

Transputer development system delivery manual

The delivery manual provides instructions for installing the software on the host computer, and checking that the installation has been successful. It also discusses the changes that have been made since the previous release of this product, and lists known problems in the software and documentation.

Transputer development system user guide and reference manual

The user guide and reference manual are provided in a single volume published by Prentice Hall (ISBN 0-13-928995-X). The user guide provides a complete introduction to the software and is designed to be read while starting to use the software. The reference manual describes the behaviour of the tools and defines the user interfaces and libraries in detail; it is supplemented by tabulations of useful information in appendices. The book is fully indexed.

occam 2 reference manual

This book, published by Prentice Hall (ISBN 0-13-629312-3), is the definitive reference manual for the occam 2 language.

Tutorial introduction to OCCAM programming

This book, published by BSP Professional Books (ISBN 0-632-01847-X), contains a tutorial introduction to concurrent programming using the OCCAM language and also a concise formal definition of the syntax and semantics of the language.

The Transputer Applications Notebook - Systems and Performance

This book is a compilation of INMOS technical notes on hardware, systems, software, applications, and performance. The notes contain discussions of important issues in transputer systems design and were written by the engineers who pioneered this work.

Software components

The software is installed in DOS directory `\tds3` and its subdirectories. The software in each subdirectory is listed below:

system

This directory includes the necessary DOS files for calling the TDS, both on IBM and NEC PCs, the TDS loader, the code of the TDS and its principal utilities, and the debugger.

complib

This directory contains the standard OCCAM libraries. This includes the support for long integer and floating point arithmetic on all processor types.

tutor

This directory contains the on-line tutorial and other examples mentioned in the user guide.

examples

This directory contains a wide variety of examples of OCCAM programs.

tools

This directory, and a source subdirectory, contain the following tools and examples:

- Lister and unlist programs
- Program for transferring TDS files over a link
- Transputer network tester
- Tool for building stand-alone hosted programs

- EPROM making tools, including the 'memory interface program'
- Example ROM sources
- Disassembler

Most of the tools are supplied as OCCAM source so that the user may modify them to local requirements if necessary.

iolibs

This directory and its subdirectories contain a wide variety of library code, mostly concerned with input and output operations. Some of this library code is specifically designed to support the run-time interface available to a program running within the TDS; some of it is useful for programs running on a separate transputer board or network.

hostlibs

This directory and its subdirectories contain library code for communication directly or indirectly with the server on the host. These procedures match those provided in the IMS D705 OCCAM toolset.

mathlibs

This directory and its subdirectories contain the mathematical libraries. These include all the usual trigonometrical functions, exponentials, and random number generators.

iserver

This directory includes all the sources necessary to regenerate the server program which can be used on the host computer to support the TDS, or programs generated using it, or by other INMOS software products.

5.6.3 Product components

Documentation

- Tutorial introduction to OCCAM programming
- OCCAM 2 reference manual
- Transputer development system delivery manual
- Transputer development system user guide and reference manual
- Applications Notebook

Integrated software components

- OCCAM 2 checker and compiler
- OCCAM 2 network configurator
- Separate compilation and library system
- Network loader
- Source-level debugger
- Network explorer and tester
- EPROM making tools

- File handling utilities
- Program loader and file server (in C)

Software libraries

- compiler libraries
- mathematical functions (sin, cos etc)
- mathematical functions optimised for fixed point processors
- string conversion and manipulation procedures
- host file and terminal i/o procedures
- evaluation board terminal support
- handling of transputer link communications

5.6.4 D705B OCCam 2 toolset support

The D705B is an alternative vehicle for the development of OCCam programs for transputers. It is a collection of portable development tools, called from a DOS command line environment. There are similar toolsets for the scientific programming languages (C, Fortran and Pascal).

A number of tools are provided with the D705B to support its use in conjunction with the D700. It is also possible to import programs and modules compiled with the D705B and scientific language compilers into the D700.

5.6.5 Operating requirements

- IBM PC or NEC PC-9801 (AT or XT compatible, 256 Kbyte memory)
- IMS B004, IMS B010 or (IMS B008 + IMS B404) transputer board or equivalent
- At least 1 Mbyte of transputer memory (greater flexibility with 2 Mbytes or more)
- DOS Version 3.0 or later
- 5 Mbytes of free hard disk space (preferably on 20 Mbyte or larger disk)

5.6.6 Distribution media

Software is distributed on 360 Kbyte (48TPI) IBM format floppy disks.

Delivery manual and floppy disks are distributed in standard A5-sized box and binder.

5.6.7 Licensing information

There is no licence fee incurred for including the binary of the INMOS libraries in software products. Example programs, and other source provided, may be included in software products. Full licensing details are available from INMOS.

The following products are also available for licensing:

D700-B D700 binary distribution release

D700-S D700 source release

5.6.8 Error reporting and field support

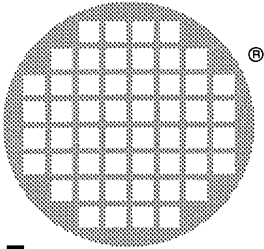
Software problem report forms are included with the software.

INMOS has a world-wide network of sales offices, providing support for INMOS software products. In some areas the functions of support may be taken over by distributors or other third parties.

Customers requiring a formal support contract should contact their local INMOS sales office.



Board Support Software



inmos

IMS S708 IMS S514 Motherboard Support Software

DESCRIPTION

The module motherboard software support packages provide full support for the INMOS module motherboards including device drivers, server programs and switch setting software.

KEY FEATURES

- Host device drivers
- Inmos server program plus sources
- Support for switch setting on INMOS motherboards
- WORM program to investigate attached Transputer networks
- Support for Inmos development system
- Support for application programs

6.1 IMS S708 and IMS S514 product overview

6.1.1 Product overview

The Motherboard Software Support packages provide full support for INMOS Motherboards.

Each package consists of a device driver, an inmos server, and the module motherboard switch setting software.

The device driver allows the motherboard to be installed as a standard device in the host machine which can be accessed through operating systems calls rather than by accessing the interface registers directly.

The board installation and installation test procedures are fully documented in the User Manual.

The module motherboard software will allow you to program your desired network topology. You provide a logical description of the boards and the connections which you require and the MMS generates a program to set the IMS C004 switches accordingly.

The motherboards provide a route for connecting networks of transputers to your host machine either by adding transputer modules (TRAMS) or by transputer links to embedded systems. The module motherboard software provides a worm program to explore the connected network and report on the configuration that it finds.

The INMOS server will load programs to the connected network and provide access to the host file system and terminal for programs running on the B014. The INMOS server is used to load and execute the module motherboard software. The INMOS server also supports execution of the Occam 2 toolset and applications built with this toolset on the B014.

Support for other hosts

The module motherboard support software provides support for specific host and board configurations. Other host and board configurations can easily be supported as the INMOS server is provided as C source. The board dependent part of the server is captured in a module called link.c which defines a standard interface for reading and writing data from a host machine to a link. Porting this simple program will allow you to run the programs supplied with this package on other host and board configurations. The design of this program and advice on how to port it to a new system is documented in the User Manual.

Having ported the INMOS server to your system you will also be able to run the Occam 2 Toolset and applications built with this toolset.

The iserver and link.c specifications are open standards maintained by INMOS. A wide range of companies support these interfaces for their own transputer boards.

6.1.2 Product components summary

Documentation

- User manual

Software tools

- Device driver
- File server and loader
- Module motherboard switch setting software

6.1.3 IMS S708

The motherboard support software for the IBM PC XT or AT is termed the IMS S708.

Operating requirements

- IBM PC XT or AT, or NEC PC-9801 with 256 Kbyte memory
- IMS B008 or IMS B015 plus a TRAM with at least 2 Mbytes (e.g. IMS B404)
- At least 1 Mbyte of transputer memory
- DOS 3.0 or later
- 1 Mbyte of free disk space

Distribution media

1 360K 48TPI IBM format floppy disk

6.1.4 IMS S514

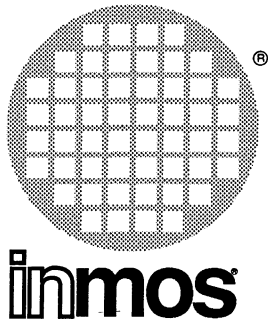
The motherboard support software for the Sun 3 workstation is termed the IMS S514.

Operating requirements

- Sun 3 Workstation
- SunOS 4.0 or later
- 1 Mbyte of free disk space
- IMS B014 VME Motherboard plus a TRAM with at least 2 Mbytes (e.g. IMS B404)

Distribution media

Sun Data Cartridge (Quarter inch tape) in tar format.



IMS S607

IMS S507

Ethernet

Support Software

KEY FEATURES

- Fast download of programs over Ethernet to transputer networks
- Support for remote debugging of programs from host terminal
- Support for shared use of transputer networks
- Support for VAX and SUN systems

6.2 Ethernet Support Software product overview

6.2.1 Product overview

The primary objective of this development is support cross development of transputer software by allowing programs to be rapidly downloaded to the target hardware over Ethernet and debugged remotely. A secondary objective is to allow transputer programs to be executed remotely from the VAX.

6.2.2 Product description

A users can attach a transputer network to Ethernet.

A user at a VAX or SUN terminal will be able to download transputer programs from a VAX over the Ethernet to the transputer network to be executed. The programs can be debugged remotely from the terminal using the standard INMOS cross development tools.

Multiple users can gain access to the transputer network from their own terminals.

6.2.3 Software components

The VAX Link software consists of two parts

- Host server
- Transputer communications software

The host server is an extended implementation of the iserver program distributed with the toolset. The server will download programs to the target transputer network and support the remote debugging of this program. The server also provides the application program with host file and terminal services. The server will be provided in source and binary forms.

The transputer communications software will be implemented as firmware on the INMOS Ethernet TRAM. This software will only be provided as binary.

Communication between the components will be achieved using TCP/IP.

6.2.4 Hardware requirements

- An EPROM module to hold the transputer software
- B407 Ethernet module
- B403 T4 plus 1Mb module
- Power supply, rack and motherboards.

The user may attach a number of transputer networks to this system and define names for each network.

6.2.5 Compatibility considerations

The implementation will be compatible with other INMOS software development products: occam 2 toolsets, 3L C and FORTRAN compilers, and Alslys Ada compiler.

6.2.6 Performance

The performance of the download system should be as good as a B008.

6.2.7 User Documentation

There will be a single user manual divided into two sections.

There will be an installation section which will explain how to set up the hardware, install the software and check that the system is operational.

There will be small user guide section which will explain how to use the server with INMOS supplied development tools.

6.2.8 Distribution media

The transputer software will be distributed on INMOS EEPROM TRAMs with the EPROM ready programmed.

The host software will be distributed on either TK50 cartridge or SUN data cartridges.

6.2.9 Related products

Source and binary distribution releases of the TCP/IP software will be made available to OEM customers wanting to build transputer based systems which can be connected to Ethernet.

PLEASE NOTE

Note: The first VAX implementation will only allow a single transputer network to be attached to the Ethernet TRAM.

Note: VAX users must have the DEC TCP/IP running on the VAX to support the server to transputer communications over Ethernet. This software is available from DEC.

Note: The first VAX product will use a restricted implementation of the UDP protocol (Assumptions will be made regarding fragmentation of messages based on the use of Ethernet), plus internally defined protocols to support INMOS subsystem RAE, and delivery checking of UDP packets.



Transputer Development Kits

7.1 Transputer Introduction Kit

The INMOS Transputer Introduction Kit provides an inexpensive entry point for the evaluation of transputers. It provides the minimum necessary configuration for the use of any of the INMOS software products. However, the motherboard allows the addition of many more modules to create a very powerful transputer network on a single board. There are two options for the introduction kit.

Kit 1 includes:-

- IBM PC Motherboard (IMS B008)
- PC device driver for the motherboard (IMS S708)
- 2 Mbyte DRAM TRAnsputer Module (TRAM) for hosting development software (IMS B404-4)
- one piece of software from:- Parallel C, PASCAL, Parallel FORTRAN, Transputer Development System or PC Toolset

Kit 2 includes:-

- NEC PC Motherboard (IMS B015)
- 2 Mbyte DRAM TRAnsputer Module (TRAM) for hosting development software (IMS B404-4)
- one piece of software from:- Parallel C, PASCAL, Parallel FORTRAN, Transputer Development System or PC Toolset

7.2 Transputer Performance Evaluation Kit

The INMOS Transputer Performance Evaluation Kit provides all the necessary hardware and software for you to evaluate the full potential of the transputer. It will typically be used for measuring real-time applications performance and benchmarking. It provides four high performance transputers each containing a Floating Point Unit and a variety of memory configurations. The Performance Evaluation Kit includes:-

- IBM PC Motherboard (IMS B008)
- PC device driver for the motherboard (IMS S708)
- IMS T800-25 + 2 Mbytes DRAM size 2 TRAM (IMS B404-5)
- IMS T801-20 + 160k SRAM size 2 TRAM (IMS B410-11)
- IMS T800-20 + 1 Mbyte size 4 TRAM (IMS B403-3)
- IMS T800-25 + 32k SRAM size 1 TRAM (IMS B401-5)
- PC Toolset (IMS D705) plus
a Parallel C or PASCAL or Parallel FORTRAN compiler.

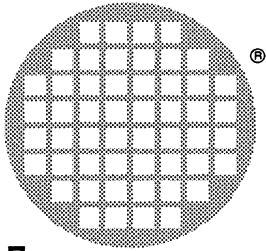
7.3 Custom Development Kits

If neither of the INMOS development kit packages precisely meets your requirements, it is very straightforward to put together your own. There are a variety of motherboards available for several host platforms, and the wide range of compatible TRAMs provides you with the freedom to tailor a system to your particular needs.

To produce a custom development kit, you need a minimum hardware configuration of an IMS B008 motherboard and a TRAM with at least 2 Mbytes of memory (eg. IMS B404, IMS B405, IMS B417). Any number of other TRAMs and motherboards can be added to this minimal system, either as part of the initial system, or

later as an upgrade. To this hardware you add the PC development tools that you require from the following:

- Parallel C compiler (IMS D711)
- PASCAL compiler (IMS D712)
- Parallel FORTRAN compiler (IMS D713)
- PC Toolset (IMS D705)
- Transputer Development System (IMS D700)



inmos

IMS B211

INMOS Transputer Evaluation Module (ITEM)

KEY FEATURES

- Slots for 10 double extended Eurocards
- Rugged built-in power supply capable of delivering 50A @ 5V and 8A @ 12V
- Built-in forced air cooling
- Removable card assembly allowing easy access
- Meets U.S. FCC standards (part 15, subpart J, limit A)

7.4 IMS B211 INMOS Transputer Evaluation Module (ITEM)

7.4.1 Introduction

The IMS B211 Transputer Board Rack (or ITEM, INMOS Transputer Evaluation Module) is a modular cabinet that has been designed to accommodate up to 10 INMOS double extended Eurocard transputer boards. For example, the B211 will accept boards such as the B003, B005, B006, B007, and the B012 motherboard. It will also accept other boards of the Eurocard format.

7.4.2 Applications

The B211 provides a simple means of connecting transputer boards together with the necessary power and cooling requirements to provide potential supercomputing power. The B211 is consequently suitable as an evaluation vehicle for large transputer systems or as a powerful embedded accelerator accessible from a host.

The B211 is designed to be upgradeable to meet the user's changing requirements as a project evolves. Further evaluation boards can be easily added to give additional functionality such as disk storage or graphics. If greater computing power is required, the multiple B211's can be linked and stacked to build even larger transputer arrays.

Several INMOS products exist which make use of the B211. One such product is the ITEM 4000 (IMS B213-4), which includes 10 transputer boards, each containing 4 IMS T800-G20 floating point transputers and 1 MByte of RAM. A Transputer Development System (IMS D701-4) for the IBM PC XT/AT is also included. This provides the user with a complete transputer development system, a performance of 800 MIPs and a sustainable processing power of 90 MFLOPS!

Other configurations are possible. For example, an ITEM complete with 10 IMS B012 motherboards has a maximum processing power of 3200 MIPs/360 MFLOPs sustainable power, when each one is loaded with 16 T800-G20 Transputer Modules or TRAMs (eg. IMS B401-3).

7.4.3 Rear Connector Panel

The connector panel at the rear of the ITEM includes:

- Four BNC connectors for linking the ITEM to colour monitors.
- Two 25-way D connectors for RS232 connection terminals, computers, and peripherals.
- Two 37-way D connectors, each of which can carry 12 transputer links, and three system service ports. A cable is supplied with the ITEM to connect to these D connectors.

7.4.4 FCC Compliance

The IMS B211 has been found to comply with the limits for a Class A computing device pursuant to subpart J of Part 15 of the FCC regulations. These rules are designed to provide a recognised standard to protect against the emission of any harmful interference to radio communications within a commercial environment.

7.4.5 Ordering Information

Description	Order Number
IMS B211 for 110V AC mains supply	IMS B211-1 US
IMS B211 for 240V AC mains supply	IMS B211-1 UK

Table 7.1 Ordering information

The facility to change the mains power requirements from 110V to 240V and vice-versa is provided.



Applications



Dual Inline Transputer Modules (TRAMs)

8 Dual Inline transputer modules (TRAMs)

8.1 Background

INMOS has built a number of transputer evaluation boards. Most are the same size (220mm x 233.4mm), which fits the INMOS ITEM. These boards have different transputer configurations and different amounts of memory (IMS T212, T414, T800, M212, transputer graphics, several transputers, 64K to 2M of RAM). INMOS has also produced boards to fit particular computers, such as the IBM PC and the NEC 9801.

The need

It would have been nice if we had been able to offer all the different transputer configurations to fit into these personal computers. But instead of about ten different designs of boards, this would have meant 30 different designs. And there was market demand for transputers to plug into VME, to VAX, to SUN, to other workstations, process control computers, minicomputers, mainframes. And there was further demand for more configurations, such as more memory per transputer, more transputers with less memory, or the same memory in much less space, graphics and other different peripherals.....

Clearly to produce all these different transputer configurations, to plug into all these different computers, would need over 100 different board designs. Even if INMOS could design those, it would be foolish to stock and sell so many different designs. But a genuine market demand existed to be met. Somehow we had to separate the transputer configuration from the computer and its size and shape of board.

Meeting the need

A small range of transputer configurations, implemented as modular subsystems, and a small range of motherboards with sockets for the modules, offered this separation.

Users can mix and match different physical sizes of modules, modules with different memory sizes and modules with different functions. By mixing and matching, many more than 100 different combinations are possible.

An advantage to many customers who have the expertise in interfacing to their own computers is that they can design their own module motherboards, and use the ready-built transputer configuration supplied as modules. This should greatly reduce the time needed to prototype a transputer system.

The building block

In effect the module is a board level transputer, with a very simple standardized interface. The building block concept is practically realized by integrating memory and peripheral functions on board, and by limiting the pin out to 16 pins (although some modules use several sets of these 16 pins). It is just as easy to build transputer circuits with modules as it used to be to build logic circuits out of TTL.

Several of the modules are densely packed, offering thousands of MIPs, hundreds of MFLOPs and many megabytes, all on a few motherboards in a small box.

Two questions

Two questions are frequently asked - why DIL, and why just this size?

We use DIL because it is more robust than SIL when assembled on the board; also because the height of a transputer SIL strip would be over 1" using PGA transputers. The pin out of adjacent modules is arranged, however, so that if at some future time SIL strips appear viable, the SIL pinout works.

The size comes from considering how small a transputer could become. As the chip is about 1cm square, it would not fit with a 0.3" 16 pin DIP, but it would fit into a 0.6" 16 pin DIP. Put four of these on a regular prototyping board with rows of sockets on 0.3" centres and you have a set of pins 9-16 just 3.3" away from pins 1-8. Add enough at each end for mechanical fixing and width for a PGA to give the final size.

So the size was primarily chosen to fit standard prototyping boards. Conveniently, the size also fits the IBM PC, VME boards, and the INMOS ITEM, as well as a host of other computers.

8.2 Introduction

TRAMs are small subassemblies of transputers (or other components with INMOS links), a few discrete components, and sometimes some RAM and/or application specific circuitry. They:

- interface to each other via INMOS links

- have a standard pinout

- come in a range of standard sizes

The basic size of a TRAM is 1.05" by 3.66" overall, about half the size of a credit card. This basic size is referred to as Size1. Larger TRAMs can be up to 8.75" by 3.66", which fits comfortably on an IBM PC board or on a VME board (this largest size is referred to as Size8). Smaller TRAMs (hybrids or silicon, not yet implemented) can be as small as a 16 pin DIP with leads on 0.6" centres.

The standard pinout and standard sizes of TRAMs make it very simple for users to build customized motherboards with sockets for TRAMs. These can either be in prototype form (Perfboard, Vectorboard or Veroboard), or in printed circuit form.

TRAMs may be plugged into the TRAM sockets on any of the following INMOS evaluation boards: B006 (eight Size1 modules), B009 (one Size4 module), B010 (four Size1 modules), and B011 (two Size1 modules). Connections between modules are hard wired on the B006 as two squares; on the other boards the links are connectable either at header plugs or at an edge connector.

The IMS B008 and B012 are specifically designed for TRAMs. Both boards can be connected into a wide variety of different networks by 'softwiring' connections between transputers by using the IMS C004 link switch. The B008 takes 10 Size1 TRAMs and plugs into the IBM PC, The B012 takes 16 Size1 TRAMs on a double extended Eurocard and plugs into the INMOS ITEM. INMOS will introduce other boards to fit other hosts.

The TRAM standards referred to above are independent of:

- transputer type (IMS T212, T414, T800, M212, etc.)

- number of transputers (1, 4, 8, 12, 16 are all possible)

- wordlength of transputer (16 bits on T212, 32 bits on T414)

- speed (T414-15, -20, to T800-30 and beyond)

- function (transputer plus RAM, disk control, other peripheral control)

- memory size (no external RAM up to many megabytes)

- package (68 pin PGA, 84 pin PGA, PLCC, and other transputer packages)

- implementation (through-hole PCB, surface mount PCB, hybrid, silicon)

Further information is available from INMOS on the B008 and B012 module motherboards, and on the product family of TRAMs.

8.3 Functional description

8.3.1 Pinout of size1 module

The pins include four INMOS links, which require no off-module buffering.

Table 1 shows the pinout. This pinout has been chosen partly to simplify layout of the motherboard, and partly to simplify the layout of the TRAM.

Table 1: Standard TRAM pinout

1	Link2out	Link3in	16
2	Link2in	Link3out	15
3	VCC	GND	14
4	Link1out	Link0in	13
5	Link1in	Link0out	12
6	LinkSpeedA	notError	11
7	LinkSpeedB	Reset	10
8	Clockin(5MHz)	Analyse	9

When LinkSpeedA and LinkSpeedB are both low, the TRAM links operate at 10Mbits/s. When they are both high, the links operate at 20Mbits/s. Other states of these pins are reserved for future enhancements.

The notError signal is driven by an open collector transistor so the signal can be wire ORred. This allows for the error line to be bussed in the same way as Clock, Reset, and Analyse. The fan-in of the notError signal must be controlled, and it is recommended that no more than ten notError outputs are wired together.

Pin 1 is marked by a silk screened triangle.

8.3.2 Pinout of larger sized modules

Figure 8.1 shows two adjacent Size1 TRAMs side by side. Notice that the orientation of the two modules is different. This difference in orientation serves two purposes: cooling of Size1 modules is improved; and it makes it possible at some future date to have Single-In-Line modules.

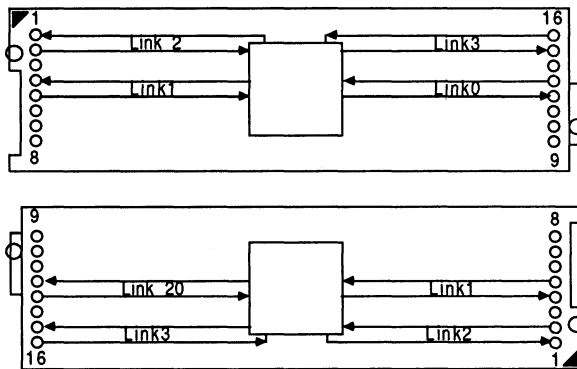


Figure 8.1 Orientation of adjacent Size1 modules

Many modules, and all the early products IMS B401 to B405, contain a single transputer, and so do not need more than one set of 16 pins for electrical signals. Modules larger than Size1, however, are assembled with extra sets of 16 pins; the extra pins give mechanical support, allow modules to be stacked, and provide extra GND and VCC pins. A Size2 module with one transputer is shown in figure 8.2a.

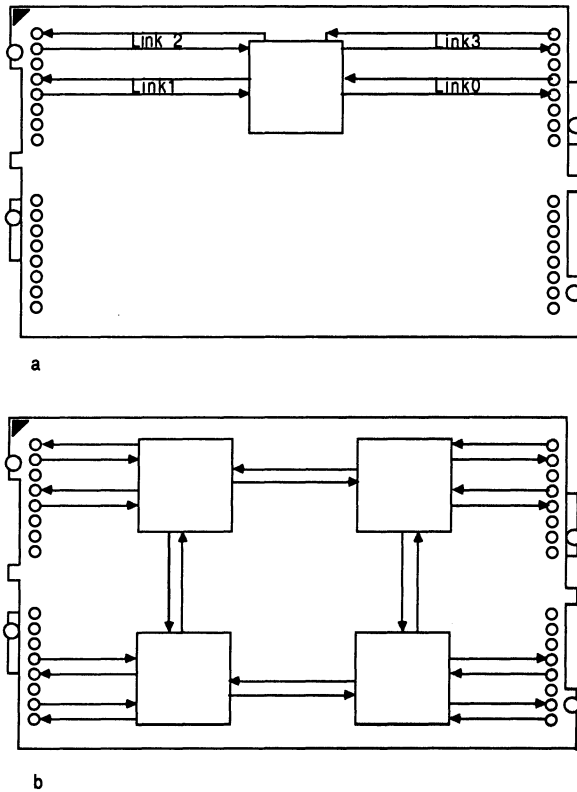


Figure 8.2 Size2 TRAMs with one and four transputers

TRAMs may be built with more than one transputer, or with transputers having more than four links. An example of a possible TRAM with more than one transputer is shown in figure 8.2b. This has four transputers connected as a square, in the same way as the IMS B003 and B006. (In practice, if INMOS were to produce a TRAM with four transputers, the links would probably be routed to make better use of standard motherboard connections.)

The detailed pinouts of larger modules are shown with the mechanical details in section 8.8 and assume that each TRAM has a single transputer, with four links.

Notice that the Size2 module and the Size4 module have the pins which are actually used at one end. The Size8 module (when it has a subsystem capability) has the pins which are used in the middle.

8.3.3 TRAMs with more than one transputer

Standards for pinout of transputers with more than one transputer are to be defined.

8.3.4 Extra pins

TRAMs may include application specific circuitry which requires pins other than the standard 16 pins. Examples are peripheral controllers or pipelines used for graphics or signal processing. The recommended connector for these is a strip of pins on 0.1" grid, such as a stripcable socket will attach to.

8.3.5 Subsystem signals driven from a TRAM

It is useful for TRAMs to be able to control a network of transputers and/or more TRAMs. Such a slave network is known as a *subsystem* of the master, and the set of control signals from the module are described as a *subsystem port*.

The subsystem port consists of three signals: **SubsystemReset** and **SubsystemAnalyse**, which enable the master to reset and analyse its subsystem; and **SubsystemnotError**, which is used to monitor the state of the error flag in the subsystem. The polarity of these signals is such that a motherboard can be built with a master TRAM controlling slave TRAMs via its subsystem port with no buffering or gating. (Note that a change of polarity may be required for a subsystem port which goes off the motherboard.)

The three subsystem signals are located on low profile sockets which are positioned 0.1" inside the standard module pins 1-3. This is illustrated by figure 8.3.

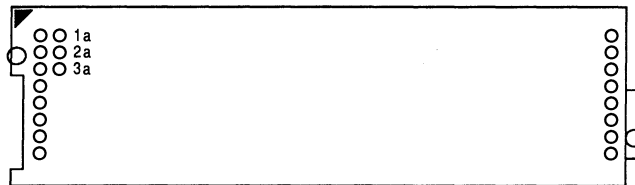


Figure 8.3 Location of subsystem sockets

The pinout is as follows:

Pin	Signal
1a	SubsystemnotError
2a	SubsystemReset
3a	SubsystemAnalyse

The sockets are fitted into the module PCB upside-down. The motherboard into which the module is plugged will also have three such sockets in the corresponding positions, but fitted from the component side in the usual fashion. The connection between the module and the motherboard is then made by a double-ended header, strip (see figure 8.4). This arrangement ensures that if the subsystem port of a module is not used, the module remains mechanically compatible with modules which do not have subsystem ports.

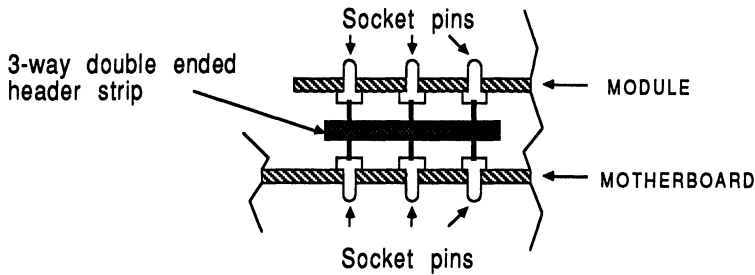


Figure 8.4 Subsystem port connections

Subsystem registers

The subsystem is controlled by reading and writing to addresses in positive address space (i.e. location zero onwards). On all INMOS evaluation boards and TRAMs, two BYTE locations are used, where each byte is the least significant byte of a 32 bit word. A further two locations control parity generation logic, which will be described in section 8.3.6. These four locations are permitted to repeat throughout the whole of the positive address space.

The subsystem registers are located at the following addresses for 32 bit transputers

Register	Hardware byte address
SubSystemResetLatch (write only)	#00000000
SubSystemAnalyseLatch (write only)	#00000004
SubSystemnotError (read only)	#00000000

The subsystem port operates as follows:

Writing a 1 into bit 0 of #80000000 asserts SUBSYSTEM Reset;
Writing a 0 into bit 0 of #80000000 deasserts SUBSYSTEM Reset.

Writing a 1 into bit 0 of #80000004 asserts SUBSYSTEM Analyse;
Writing a 0 into bit 0 of #80000004 deasserts SUBSYSTEM Analyse.

A 1 read from bit 0 of #80000000 indicates that SUBSYSTEM Error is TRUE.
A 0 read from bit 0 of #80000000 indicates that SUBSYSTEM Error is FALSE.

The subsystem is reset or analysed under the control of the transputer on the TRAM, but must also be reset when the TRAM itself is reset. To pass the signals on to the subsystem, the following combinational logic is included:

SubsystemReset = Reset OR SubsystemResetLatch
SubsystemAnalyse = Analyse OR SubsystemAnalyseLatch
the latches are initialized at power-on to be inactive.

Note that SubsystemError does NOT propagate to the TRAM's notError pin.

Multiple subsystems

TRAMs may contain more than one subsystem port. They should have their locations separated by 16 bytes.

8.3.6 Memory parity

TRAMs may include parity logic for external RAM. The implementation on TRAMs must ensure that there is no way that corrupt data can reach any other transputer.

One way to achieve this is that if a parity error occurs, the wait signal is held active so the memory cycle does not complete. All data in memory is lost, however, when an error occurs, and the memory cycle is slowed down by the parity check.

Parity checking may be enabled or disabled by writing to a parity control register. If parity is enabled and an error occurs, the error is ORed in to the notError signal from the module. Information on the cause of the error can be found by examining the parity status register.

Reset disables parity checking and deasserts MemWait. When the transputer is analysed, MemWait is deasserted and the contents of the parity status register are preserved.

The parity registers are as follows:

Register	Hardware byte address
Parity control (write only)	#00000008
Parity status (read only)	#00000008

The locations are used as described below:

Writing a 1 into bit 0 of #80000008 enables parity error detection;

Writing a 0 into bit 0 of #80000008 disables parity.

Reading the contents #80000008 returns the status of the parity detection hardware.

Bit	Status
Bit 0	Indicates a parity error has occurred.
Bits 1 & 2	Indicate the BYTE in which the error occurred. (Bit 1 is lsb).
Bits 3..n	Indicate the BANK in which the error occurred. (Bit 3 is lsb).

8.3.7 Memory map

The memory map should be of the form:

ROM	top of memory
Peripherals	
Subsystems	
External RAM	
On-chip RAM	bottom of memory

In the particular case of TRAMs with 32 bit transputers, the memory map should be as follows:

Byte address	Description	Comment
7FFF FFFF		Bootstrap program requires ROM at top of memory.
7FFF FFFE	Boot from ROM	7FFF FFFE will contain a backward jump to the bootstrap.
	Peripherals	If used
0000 000C		These locations must be decoded as a set of four, even if Parity is not used.
0000 0008	Parity status and control	
0000 0004	SubsystemAnalyseLatch	
0000 0000	SubsystemResetLatch	
8FFF FFFF	RAM	Both internal and external RAM
Memstart	RAM	

Substantial logic can often be saved by not fully decoding the hardware address. An effect of not fully decoding the address is that hardware can appear at multiple addresses.

In particular, if the module does not have a subsystem, the RAM can repeat throughout the address space, including the positive address space (above location 0).

The Subsystem and parity locations can also repeat throughout the positive address space.

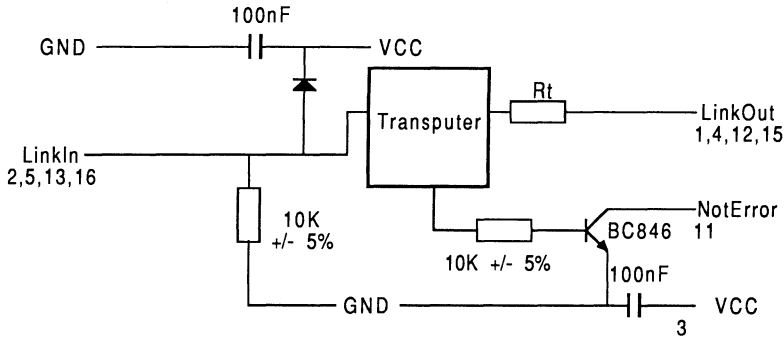


Figure 8.5 Recommended circuit between TRAM pins and transputer

8.4 Electrical description

8.4.1 Link outputs

Link outputs must be terminated so that the combined output impedance of the transputer plus termination resistors is $100\ \text{ohms} \pm 20\%$. For the optimum value of resistor, see the appropriate transputer data sheet.

8.4.2 Link inputs

Link inputs may be taken off a module motherboard and so must be protected from positive ESD by a diode to VCC. Signal diodes such as 1N4148 or LL4148 may be used. To prevent an unconnected link input from floating high, link inputs must be pulled down to GND by a resistor, preferred value $10\text{K} \pm 5\%$.

8.4.3 notError output

The notError output is a wired OR signal driven by an open collector or an open drain. Maximum leakage should not exceed 10 microamps. Maximum saturation voltage when the transistor is ON and is sinking 10 mA should not exceed 0.4 V. A suitable transistor is BC846 (SOT23) with a 10K resistor between the transputer's Error pin and the transistor base. The pullup resistor on the module motherboard should draw between 5mA and 10mA when a transistor is ON.

Although the above is conservative and should allow a fan-in of several hundred, it is recommended that the fan-in is limited to 10.

8.4.4 Reset and analyse inputs

These signals are connected directly from the TRAM pins to the transputer. They must always be driven by buffers on the module motherboard. Because the motherboard will often have filters on the Reset and Analyse signals, the Reset pulse width should be much wider than specified for the transputer. Recommended pulse width is 5 ms, with a delay of 5 ms before sending anything down a link.

8.4.5 Clock input

The TRAM must not present excessive capacitance to the clock input signal. The clock input should therefore be limited to a single load, which should be connected to the TRAM pin by a trace no longer than 30mm.

Particular care should be taken on the module motherboard to ensure that the clock input is clean, with fast edges, minimal undershoot, and minimal jitter (see transputer data sheet for clock specification).

8.4.6 notError input to subsystem

The notError input should not have a pullup resistor on the TRAM. The pullup resistor must be on the motherboard.

8.4.7 GND, VCC

Adequate high frequency decoupling capacitors must be used. In particular there should be decoupling capacitors close to the GND pin and to the VCC pin of each TRAM. Recommended value is 100 nF, preferably at least half as many as the module has ICs.

8.5 Mechanical description

In the following, dimensions are quoted in inches for PCB length, width and related dimensions; all other dimensions are quoted in millimetres.

8.5.1 Width and length

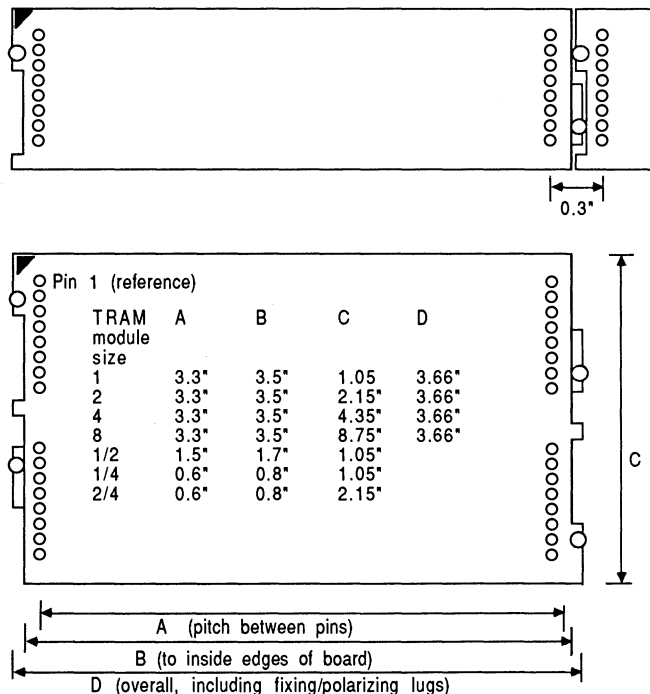


Figure 8.6 TRAM sizes

The basic size of a TRAM is a very wide 16 pin DIP, with 3.3" between the two rows of pins. These TRAMs fit on a 3.6" pitch on their length, and a 1.1" pitch on their width. Extra length is added beyond the pins to hold the pins, to provide for mechanical fixing, and to polarise the module shape.

TRAMs can be made larger than the standard size by keeping the 3.3" between pins and using two or more sets of the 16 pins.

TRAMs can be made smaller than the standard size, down to a 16 pin DIP with 0.6" between the two rows of pins, or 1.5" between the pins. These sizes will normally be used for single chip modules or hybrids.

In general the printed circuit TRAMs are longer than the pitch between the two rows of pins. The TRAMs are also wider than the 0.8" suggested by 16 pins. The small TRAMs may be side-brazed DIPs, as short as 0.8" long.

The top drawing in figure 8.6 shows a Size1 module and how the jigsaw pattern fits together between adjacent modules. The lower drawing in figure 8.6 shows the various sizes of TRAM. Detailed dimensions of the different sizes are given in section 8.8.

8.5.2 Vertical dimensions

There are no vertical height constraints for TRAMs. However, keeping the height of a TRAM, both below and above the board, within certain limits allows the TRAM to fit together with other TRAMs and motherboards.

Figure 8.7a shows height specifications which allow double-stacking of the TRAMs and which will allow two-deep stacked TRAMs on a motherboard to fit into a 1.0" pitch card-cage, (see figure 8.7e). Figure 8.7b shows how this vertical size fits onto a motherboard which has no components under the TRAM. Figure 8.7c shows the same TRAM fitted above components on a motherboard, using spacer socket strips to gain extra height.

Figure 8.7d shows another height specification which allows components such as zip packaged ICs and SMB connectors to be used on the TRAM, whilst permitting these TRAMs to fit onto motherboards in a 0.8" pitch card cage. Note that this is only possible when there are no components under the TRAM on the motherboard.

It is recommended that any component reaching a maximum specified height has an insulating surface.

Note that the datum for component heights on both sides of the TRAM is the component side surface. This datum is also used for the stackable socket to minimize tolerance buildup.

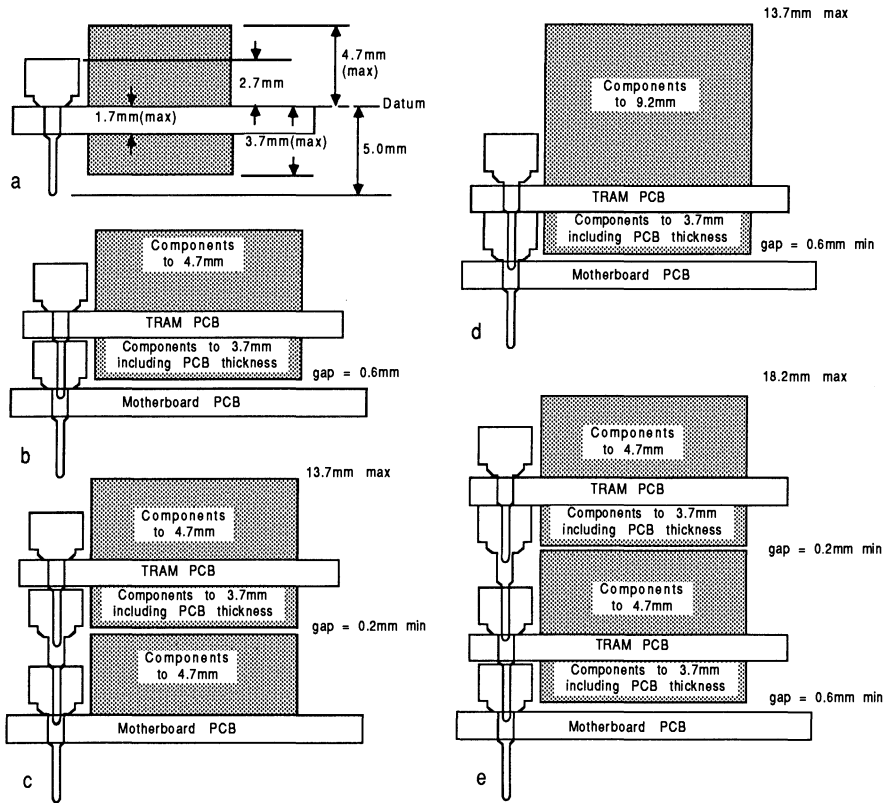
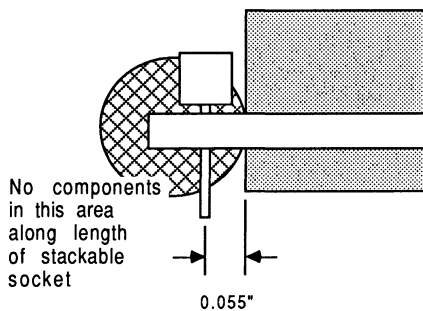


Figure 8.7 Component heights

Components must not interfere with the TRAM pins, and so the area shown in figure 8.8 must be left free of components.



No components
in this area
along length
of stackable
socket

0.055"

Components may
be placed in the
cross hatched area
between stackable
sockets, indeed this
is a suitable place
for tantalum
decoupling
capacitors.

Figure 8.8 Area close to TRAM pins

INMOS has seen and used the following sockets. No particular recommendation for any of these is given or implied. Other manufacturers have shown data sheets for similar sockets with a height of approximately 0.8mm. The Augat 'Holtite' sockets, which sit below the PCB surface, have been seen but not used. The Augat 'Solderite' sockets have similar dimensions to the Harwin 3153 and have been seen in prototype quantities. All of the sockets are available individually or assembled into strips; some are available in DIP and PGA format.

Manufacturer	type	height above PCB
Harwin (UK)	H 3153-01	0.38mm
Mark Eyelet (AMP) (US)	M8043PEC	0.2mm approx
PreciDIP (Switzerland)	014-92-001-41-012	0.4mm
Advanced Interconnections (US)	type -85	0.78mm
Harwin (UK)	H 3155-01	1.2mm
PreciDIP (Switzerland)	type 1407	0.8mm

8.6.3 Subsystem pins and sockets

The preferred socket to fit on the solder side of the TRAM is Harwin H 3153-01, and on the motherboard also. Samtec pin strip HLT-03-G-R is suitable for connecting between these sockets.

8.6.4 Motherboard sockets

The TRAM pins/stackable sockets will plug into any standard IC socket. To meet the component heights given in figure 8.7, the stackable socket (see section 8.6.1) must also be used on the motherboard.

Motherboard sockets for the Subsystem signals should be the 0.38mm or 0.4mm sockets referred to above.

8.7 Mechanical retention of TRAMs

Vibration tests have shown that in a normal office or laboratory environment, the TRAMs remain plugged into their sockets. In transit, however, or in an environment where there is vibration, some form of mechanical retention may be necessary.

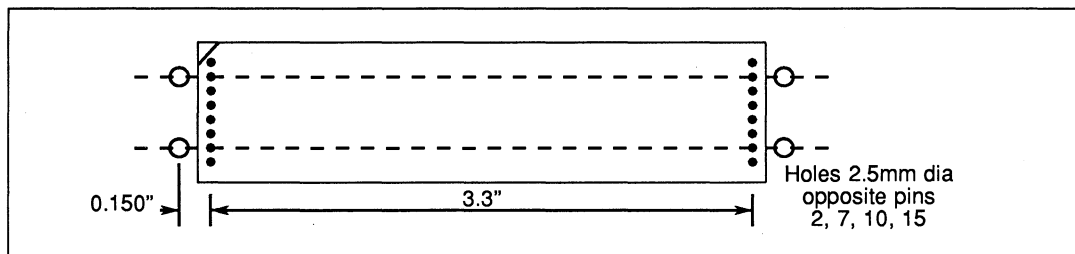
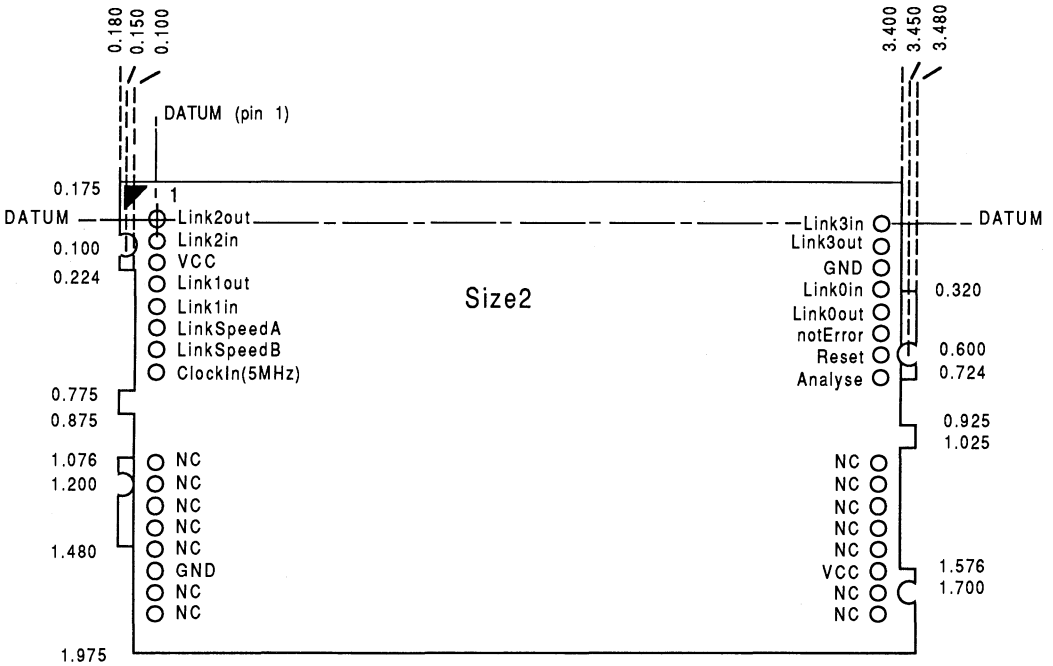
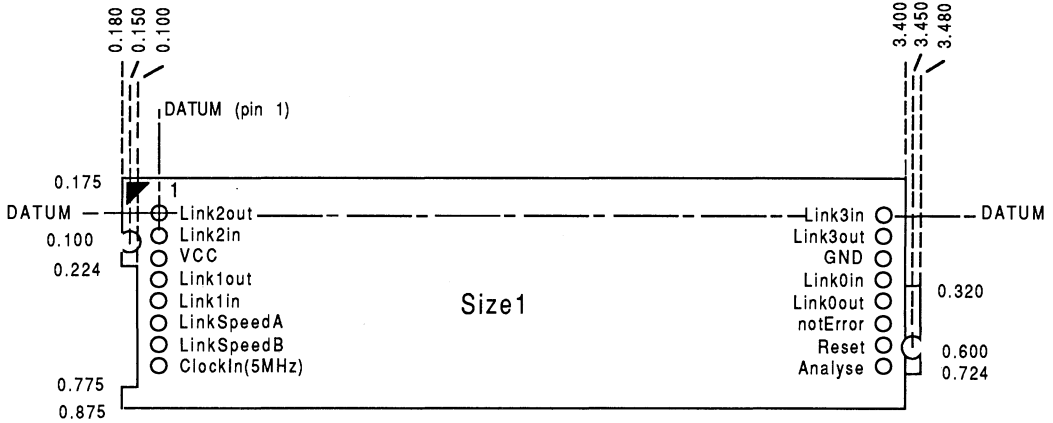


Figure 8.10 Fixing holes for mechanical retention

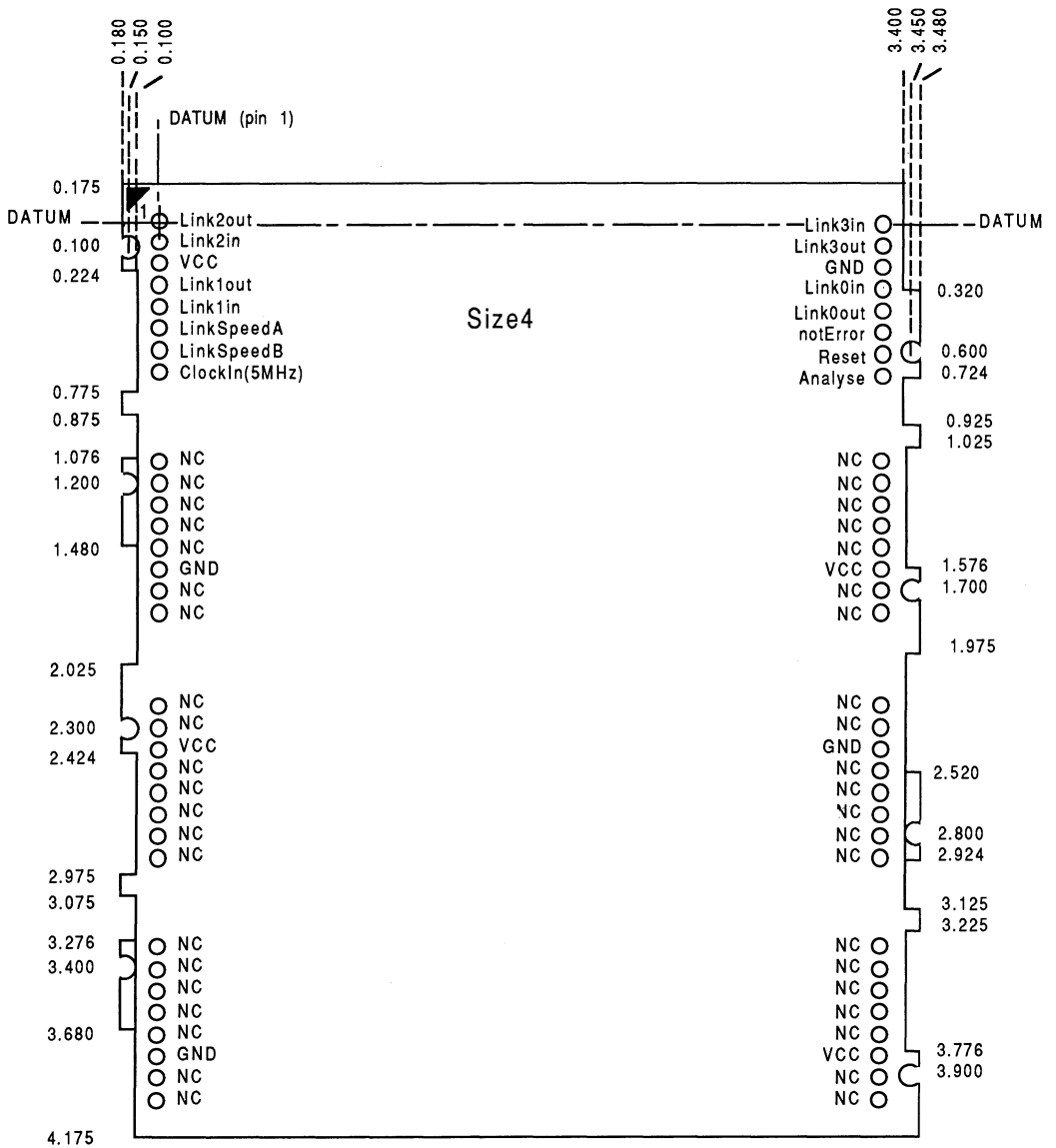
The detail drawings of the module sizes in section 8.8 show fixing holes in the modules. Similar fixing holes should be drilled in the motherboard as shown in figure 8.10. M2.5 nylon bolts may be used between these fixing holes to secure the modules.

8.8 Profile drawings



Note. All dimensions are in inches and measured from the datum line

Figure 8.11 PCB profile drawings and pinout, TRAMs Sizes 1 and 2



Note. All dimensions are in inches and measured from the datum line

Figure 8.12 PCB profile drawings and pinout, TRAMs Size 4

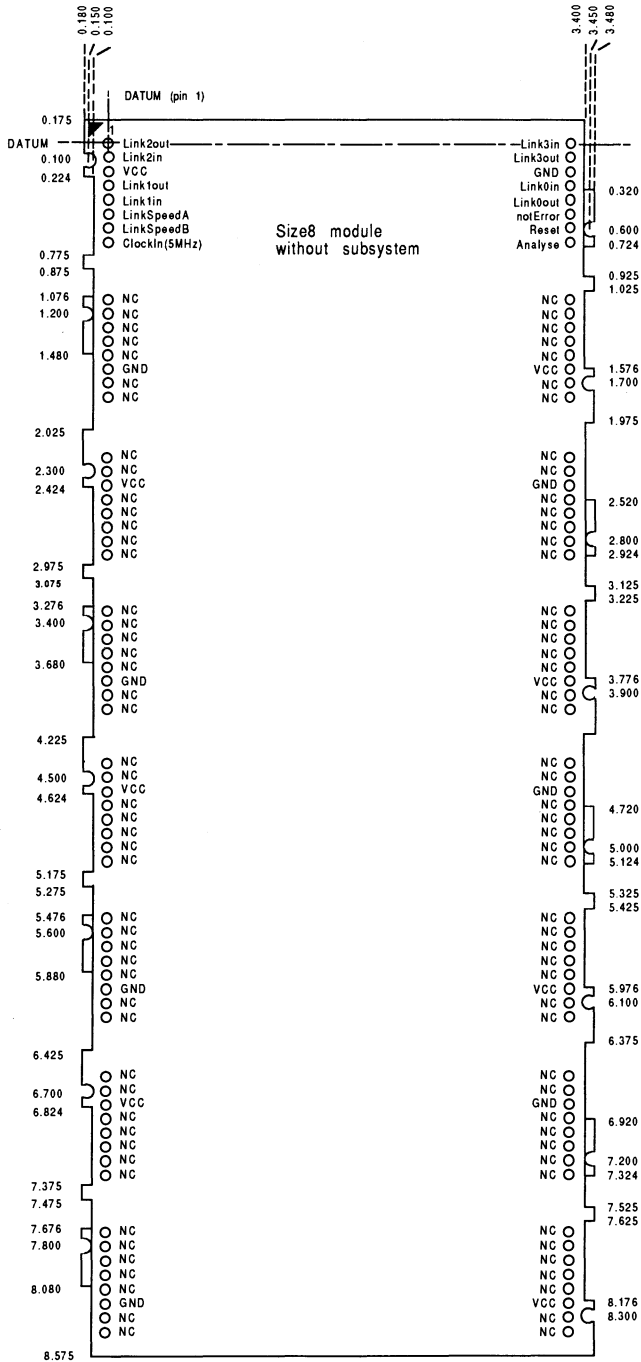


Figure 8.13 PCB profile drawing and pinout, TRAMs Size8 without subsystem



Module Motherboard Architecture

9 Module motherboard architecture

9.1 Introduction

INMOS transputer modules are designed to form the building blocks of parallel processing systems. They consist of printed circuit boards in a range of sizes which typically hold a member of the transputer family of processors, some memory and perhaps some application specific circuitry. A module needs only a 5 volt power supply and a 5MHz clock to operate. These are supplied to the module through pins on the periphery of the board. Other pins bring out the transputer's serial links and reset, analyse and error signals. Some modules can control a subsystem of other modules through another set of pins. The *Dual-In-Line Transputer Modules (TRAMs)* document provides a complete specification of INMOS transputer modules.

In order to use modules as parallel processing building blocks INMOS has developed a range of motherboards. While these boards provide access to transputers from a number of different host machines, they have a common architecture to allow control and interconnection of potentially large numbers of transputers. This document describes the generic architecture of module motherboards. It is recommended that this specification is followed when designing in order to preserve compatibility with INMOS module motherboards.

9.2 Module motherboard architecture

The INMOS range of module motherboards has a common architecture making it easy to build and configure systems consisting of large numbers of transputer modules. The goals aimed at in the design of the module motherboards, and the architecture developed to achieve them, are described below.

9.2.1 Design goals

The main goals aimed at in the design of module motherboards are:

- To be able to build systems with any number of transputer modules in any combination of type or size
- To be able to build a variety of different kinds of network (e.g. arrays, trees, cubes, etc.)
- Enable any number of motherboards to be chained together
- Make transputer link connections easily configurable by software
- To be able to run test and applications programs on transputers without first configuring links
- Provide a standard hardware interface to configuration and applications software
- Allow hierarchical control of systems of transputers
- Make the transputer hardware and software independent of the host system

9.2.2 Architecture

In order to achieve the design goals outlined above, a standard architecture is adopted for all module motherboards. The rest of this document describes the motherboard architecture in detail, but the salient features are given below.

- The modules in a network are connected in a pipeline using two links from each module
- The remaining links from each module are taken to IMS C004 programmable link switches
- A number of links are taken from IMS C004s to edge connectors for wiring to other boards

- Each IMS C004 is controlled by an IMS T212 transputer
- The IMS T212s are connected in a separate pipeline
- The first module in the pipeline on a particular motherboard can control a subsystem of other transputers that may reside on the same motherboard, another motherboard or may be distributed across a number of boards
- An interface may be provided to enable a non-transputer based host system to control and communicate with a motherboard

9.3 Link configuration

Transputers communicate with each other via serial links operating at 10 or 20Mbits/s. The module motherboard architecture facilitates the interconnection of links between transputer modules by providing a standard hardware link configuration and allowing software configuration using IMS C004 programmable link switches. Links should be interconnected by properly terminated transmission lines (PCB trace or cable) having a characteristic impedance of 100Ω. INMOS Technical note 18, *Connecting INMOS links*, gives full details on all aspects of connecting links.

9.3.1 Pipeline

Each module resides in a *module slot* which provides two sockets that take the 16 pins of a size 1 module. A motherboard may have any number of module slots, determined only by the physical size of the board. The slots are numbered starting at slot 0.

All the modules on a motherboard are connected in a pipeline as shown in figure 9.1. Link 2 of the module in

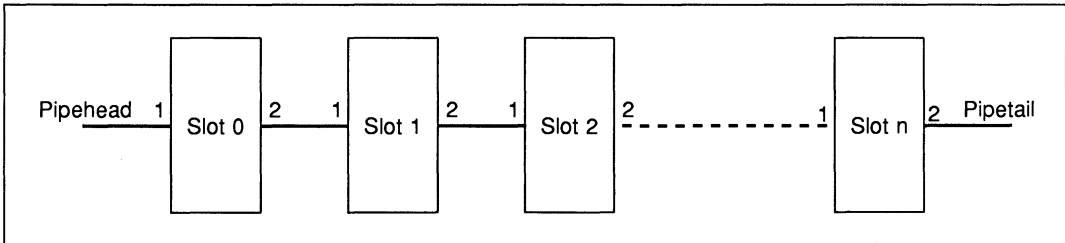


Figure 9.1 Module pipeline

slot 0 is connected to link 1 of slot 1 and so on for the rest of the pipeline. Link 1 of module slot 0 (*Pipehead*) and link 2 of the last module slot (*Pipetail*) are brought out to an edge connector thus enabling the pipelines of any number of boards to be chained together by connecting *Pipehead* of one board to *Pipetail* of the next. See figure 9.2.

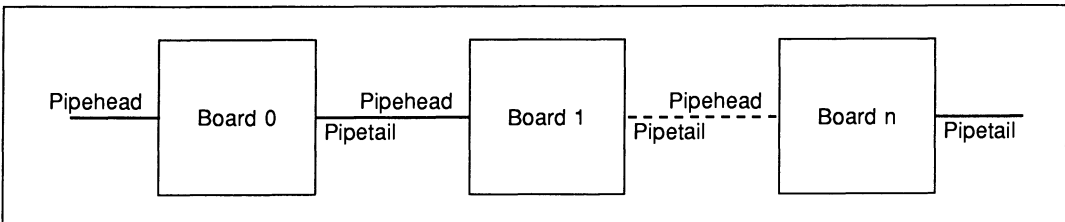


Figure 9.2 Module pipeline on several boards

Some applications may not require a full complement of modules or may use size 2 or larger modules which take up more than one slot, but use only one slot for electrical connection. In either case the pipeline will be broken unless steps are taken to keep it intact. A *pipe jumper* is a small connector used for this purpose. See figure 9.3. It plugs into an unused module slot and connects link 1 of that slot to link 2 of the same slot, thus preserving the pipeline.

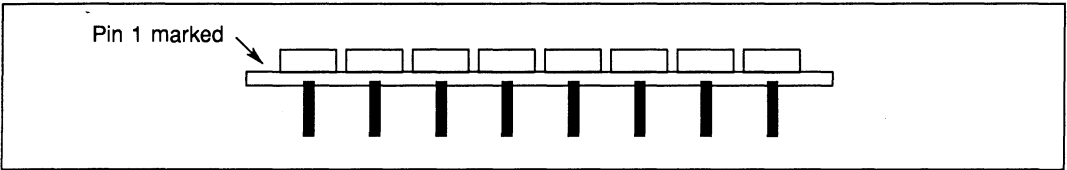


Figure 9.3 Pipe jumper

9.3.2 IMS C004 link configuration

An IMS C004 programmable link switch is used for software configuration of links. This device is a crossbar switch which can handle up to 32 links. It can connect any of the 32 link inputs to any of the 32 link outputs under software control from a separate configuration link.

Links 0 and 3 of each module are taken to an IMS C004 or a number of IMS C004s, depending on the number of links. Links may be taken from an IMS C004 to an edge connector to allow links from one motherboard to be connected to those of another.

The number of IMS C004s required on a particular motherboard depends on the number of modules the board can hold. The exact arrangement of IMS C004 links is not specified here in order to give the designer maximum flexibility for his particular application. **The only restriction is that links 0 and 3 of each module are taken to a C004.** This may be done in a number of ways. For example:

- Link 0s may be taken to one IMS C004 or a set of IMS C004s; link 3s may be taken to another IMS C004 or a set of them
- Both Link 0s and link 3s may be taken to the same IMS C004(s)
- LinkOut0s and LinkOut3s may be connected to an IMS C004 or a set of the same, while LinkIn0s and LinkIn3s are taken to another IMS C004 or a set of them

9.3.3 T212 pipeline and C004 control

Each IMS C004 on a motherboard is controlled from an IMS T212 16-bit transputer as shown in figure 9.4. An IMS T212 can control up to two IMS C004s via its links 0 and 3. Links 1 and 2 of each IMS T212 are used to connect the transputers in a *configuration pipeline*. Link 1 of the first IMS T212 on the board is taken to an edge connector designated *ConfigUp*; link 2 of the last IMS T212 in the board's configuration pipeline is also taken to an edge connector designated *ConfigDown*. In this way the configuration pipelines of any number of motherboards may be chained together by connecting *ConfigDown* of one board to *ConfigUp* of the next, enabling a network of transputer modules spread over several boards to be configured from software.

The IMS C004 configuration data may come from software running on a module residing on the first motherboard in the system. It is therefore necessary to be able to connect a link of that module to the board's configuration pipeline. A jumper provides the option of connecting link 1 of the first IMS T212 in the configuration pipeline *either* to *ConfigUp* *or* to link 1 of module slot 0. In the latter, the jumper also disconnects *PipeHead* on the edge connector from slot 0 link 1. This is shown diagrammatically in figure 9.5.

9.3.4 Software link configuration

The hardware configuration described in Sections 9.3.2 and 9.3.3 provides the standard architecture recognised by the *Module Motherboard Software (MMS)*, a software package available from INMOS which allows

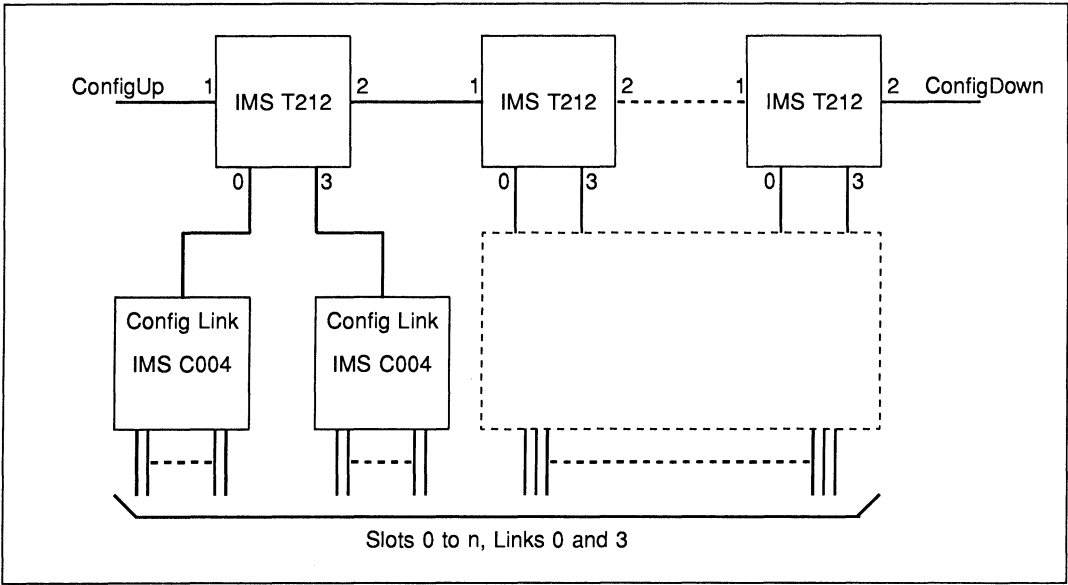


Figure 9.4 IMS C004 control by a pipeline of IMS T212s

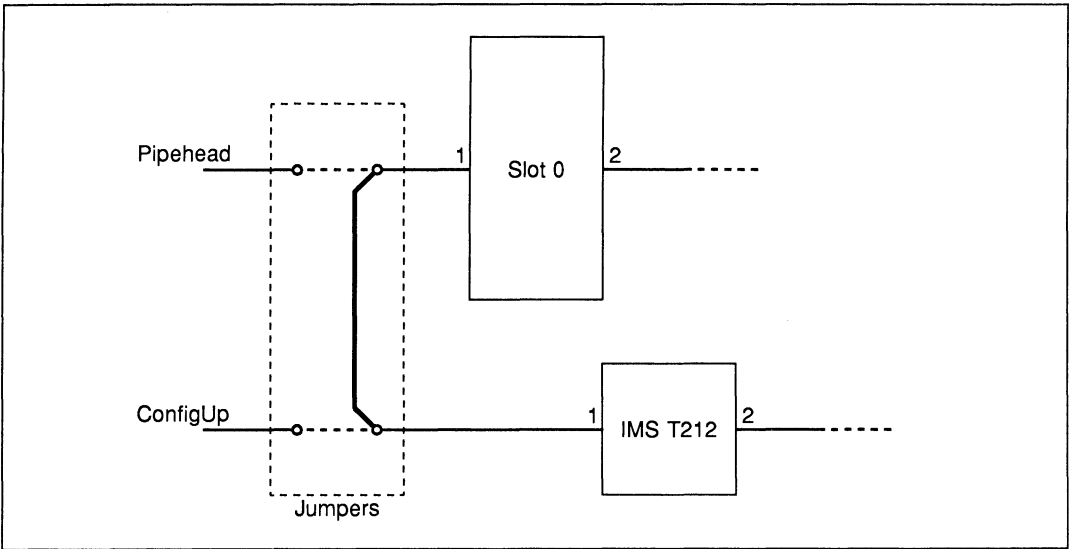


Figure 9.5 ConfigUp/Pipehead jumper

easy configuration of the IMS C004 link connections.

The MMS takes a list of link connections that are hardwired on the board together with a list of the required 'softwired' connections and generates the configuration details for each IMS C004.

For each board in the system, the user can:

- Connect link 0 of any module to link 3 of any module
- Connect link 0 or link 3 of any module to an edge connector link
- Connect an edge connector link to another edge connector link

The MMS is described in detail in the *MMS2 User Guide*.

9.4 System control

The subsystem control function of the module motherboard architecture allows hierarchical control of networks of transputers. It enables a module capable of driving a subsystem to reset or analyse a network of modules and to handle errors in the network. The driving module can itself form part of a network which is controlled by another module. In this way a hierarchy of control is made possible.

Each module on a motherboard requires a 5MHz clock. The module motherboard specification provides a scheme for distributing the clock signal from a single crystal oscillator to all the modules on a motherboard.

9.4.1 Reset, analyse and error

Three signals are provided by transputers for the purpose of allowing system control: *Reset*, *Analyse* and *Error*. The **Reset** and **Analyse** inputs enable the transputer to be initialised or halted in a way which preserves its state for subsequent analysis. The transputer **Error** signal is connected directly to the processor's Error flag. See the *Transputer Reference Manual* for a detailed description of these signals.

A transputer module has a similar set of signals: module **Reset** and **Analyse** are connected directly to the respective pins on the transputer; the transputer **Error** pin is taken to a transistor on the module to produce an open collector **notError** signal that can be wire-ORed with the **notError** signals of other modules.

Some modules are capable of controlling a subsystem of other modules. They have three extra pins: **SubSystemReset**, **SubSystemAnalyse** and **notSubSystemError**, which are controlled by the on-module transputer through latches in memory. These pins are connected to the **Reset**, **Analyse** and **notError** pins of the modules in the subsystem being controlled. The subsystem can then be reset or analysed by asserting the relevant signal of the subsystem controller module. The subsystem's ORed **notError** signal can also be monitored by the controlling module.

9.4.2 Up, down and subsystem

A module motherboard has three ports that provide hierarchical control: Up, Down and subsystem (see figure 9.6). Each port appears at an edge connector and has three active-low signals: **notReset**, **notAnalyse** and **notError**. A board is able to control a subsystem of other boards by connecting its subsystem port to the Up port of the next board. Boards in a subsystem are chained together by connecting the Down port of one board to the Up port of the next board. A board within a subsystem is in turn able to control another network through its subsystem port.

Figure 9.7 shows how a board can be connected to a subsystem of boards.

The **notReset** and **notAnalyse** signals flow from subsystem of one board to Up of the next board. From there, they go directly to Down. They are also logical ORed with that board's subsystem reset and analyse latches and then pass to the subsystem port. The **notError** signal passes from a board through its Up port. If it is connected to the Down port of the board above, it is logical ORed with that board's Error signal and passed to the Up port. If it goes to the subsystem port of the board above, the Error signal is not passed on, but is handled by that board. (Figures 9.10, 9.11 and 9.12 show the module motherboard system control logic.)

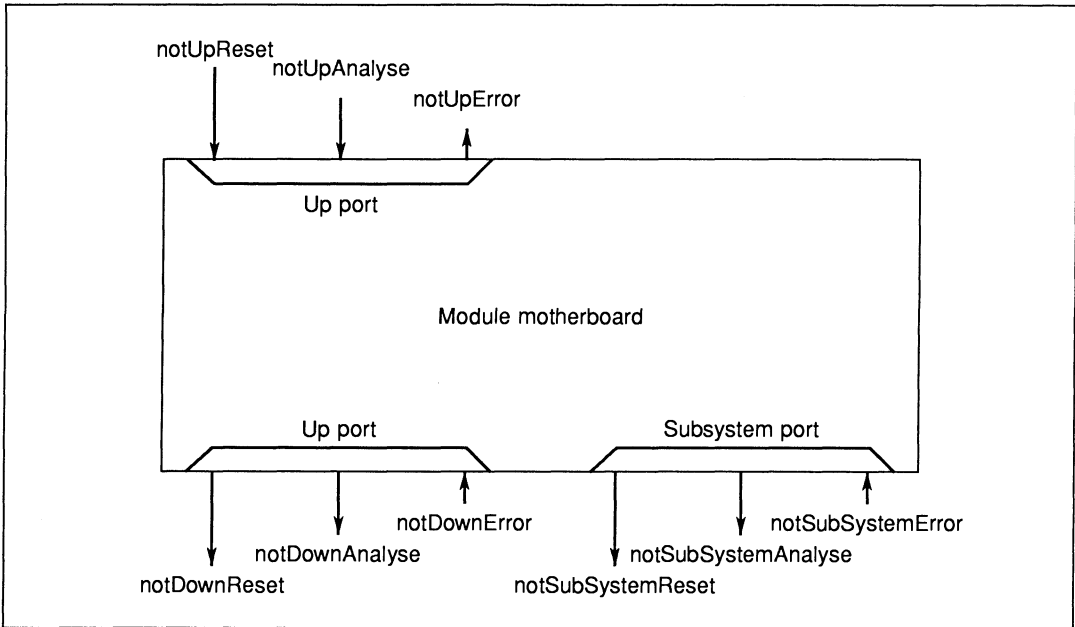


Figure 9.6 Up, down and subsystem

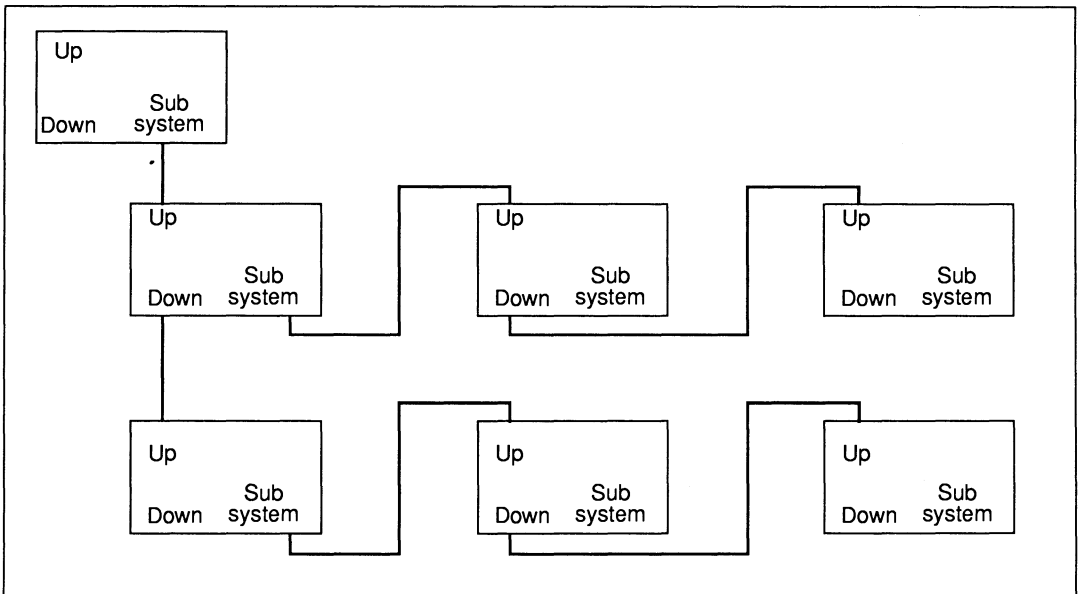


Figure 9.7 Controlling a subsystem of boards

9.4.3 Source of control

If there are n slots on a motherboard, modules in slots 1 to n may be controlled from either the Up port (or a host machine if the motherboard has an interface to one, see Section 9.5) or may be part of a subsystem controlled by a suitable module in slot 0. The source of control is determined by a jumper or switch, as shown in figure 9.8.

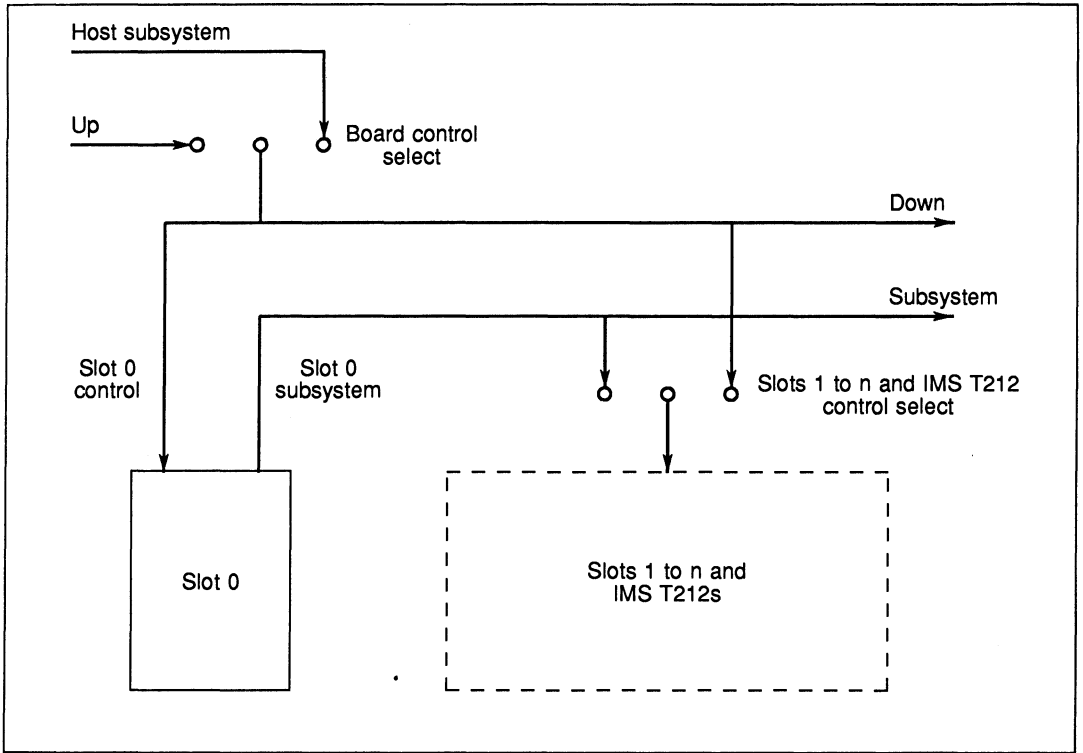


Figure 9.8 Source of control

The on-board IMS T212(s) may be reset and analysed from the same source that controls slots 1 to n . The **Error** pin of the IMS T212(s) is not connected.

A power-on reset circuit is required for the IMS C004(s) on board. An IMS C004 may then be reset at power-on or by the IMS T212 controlling it. Each IMS T212 has a latch mapped into its memory space. See figure 9.9. This enables software running on the IMS T212 to reset the IMS C004 either by setting the latch or by sending a reset message to the IMS C004 Configuration link.

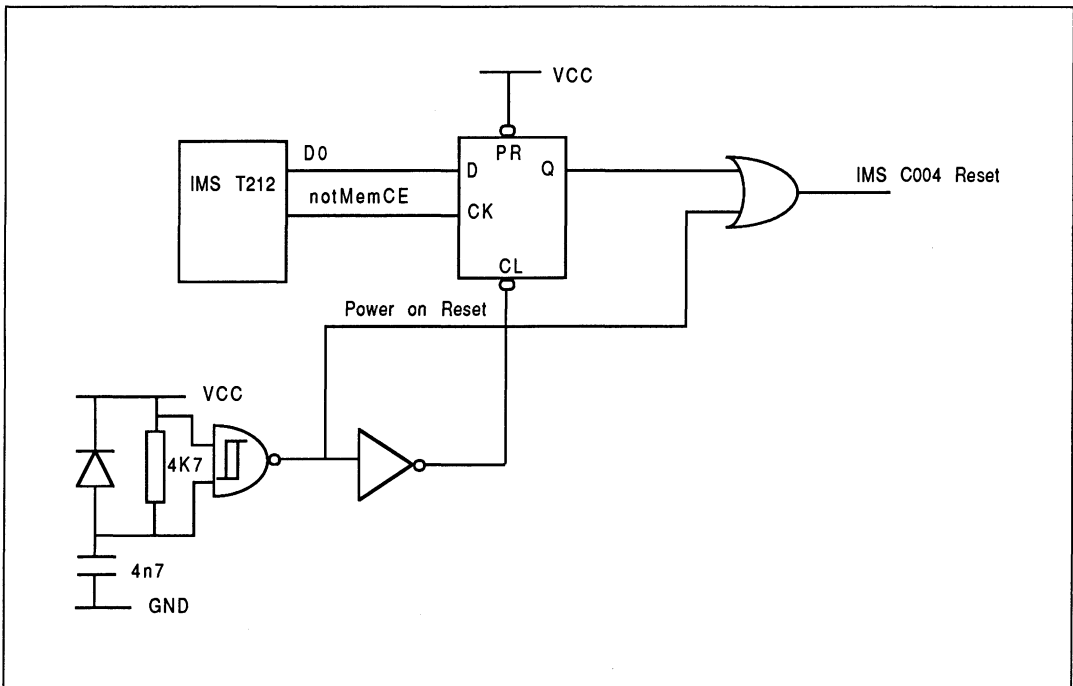


Figure 9.9 IMS C004 reset circuit

Figures 9.10, 9.11 and 9.12 show the logic required for Reset, IMS C004 Reset, Analyse and Error, respectively. These diagrams provide a logical description only: the actual implementation is left to the individual designer. It is important, however, to include the passive components indicated in the diagrams. The 1K pull-up resistors on the **notUpReset**, **notUpAnalyse**, **notDownError** and **notSubSystemError** signals are necessary to ensure that if these signals are unconnected they are not left floating, but are deasserted. The 4K7 pull-up resistors are required to wire-OR the open collector **notError** signals from the module slots. Note that the *Dual-In-Line Transputer Modules (TRAMs)* document specifies a maximum of ten **notError** signals should be wire-ORed together. The combination of each 100 Ω resistor and 100nF capacitor filters out noise on the **notUpReset**, **notUpAnalyse**, **notDownError** and **notSubSystemError** signals coming from off the board.

To improve noise rejection, it is recommended that Schmitt gates are used to receive signals from other boards. These gates should use bipolar technology (e.g. low power Schottky 74LS series TTL). It is also recommended that gates driving signals off the board are capable of providing a full output voltage swing from 0V to 5V, e.g. HCT series gates.

The Reset logic (figure 9.10) uses the **Board Control Select** switch and multiplexer to select whether Slot 0 and the Down port are reset from the Up port or from the host. The **Slots 1 to n & IMS T212 Control Select** switch and multiplexer determine whether Slots 1 to n and the IMS T212s are reset from the Slot 0 subsystem port or from the Up port or the host. A similar arrangement is used for the Analyse logic (figure 9.11).

In the Error logic (figure 9.12), the **Slots 1 to n & IMS T212 Control Select** switches and multiplexers select whether **notError** from Slots 1 to n is passed either to the Slot 0 subsystem port or to the Up port or the host. The **Board Control Select** switch and decoder determine whether **Slots 1 to n notError**, **notDownError** or **notSlot0Error** are passed to the Up port or to the host.

Board Control Select and **Slots 1 to n & IMS T212 Control Select** correspond to the conceptual switches in figure 9.8.

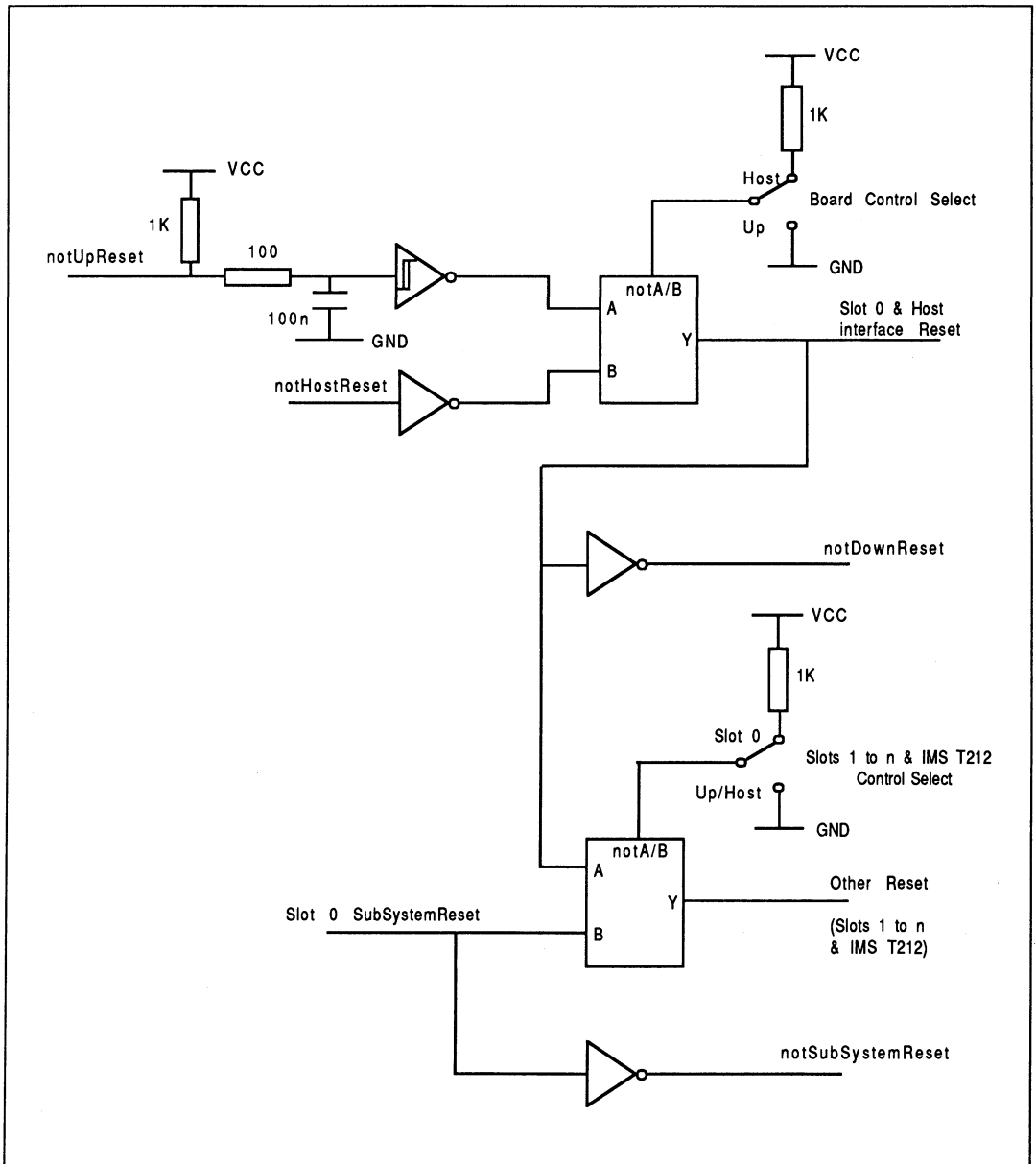


Figure 9.10 Reset logic

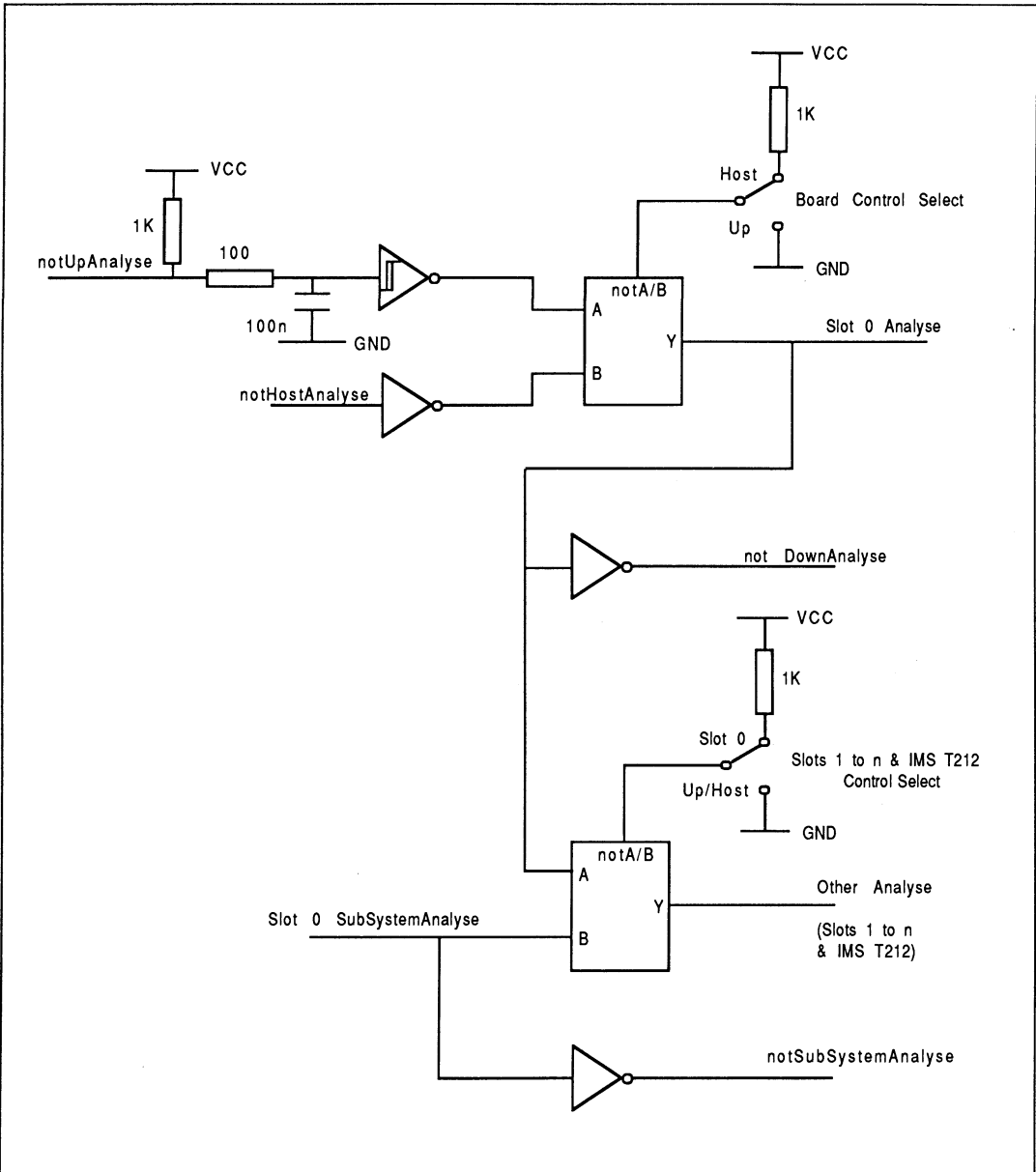


Figure 9.11 Analyse logic

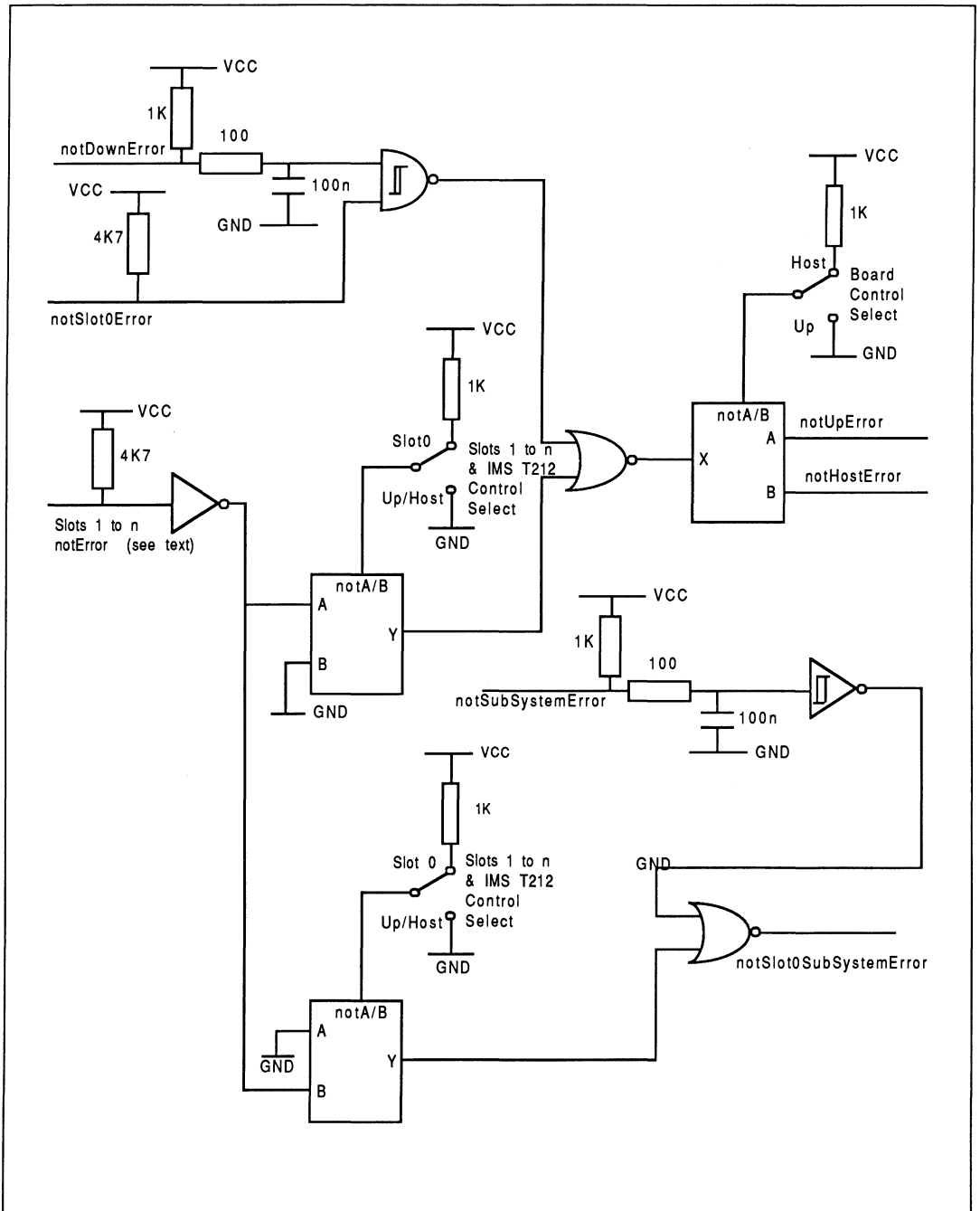


Figure 9.12 Error logic

9.4.4 Clock

A 5MHz, TTL compatible clock signal is required for each module slot, IMS T212 and IMS C004 on board. Since the clock must be distributed to a number of modules and devices the buffering scheme shown in figure 9.13 is used to minimise distortion of the clock waveform caused by excessive loading and transmission line effects. This is a *star* configuration and it may be extended indefinitely by adding more buffers at the star points which may drive further buffers, and so on until the required number of clock signals are derived. The length of any pcb trace carrying a clock signal should be limited to 30cm.

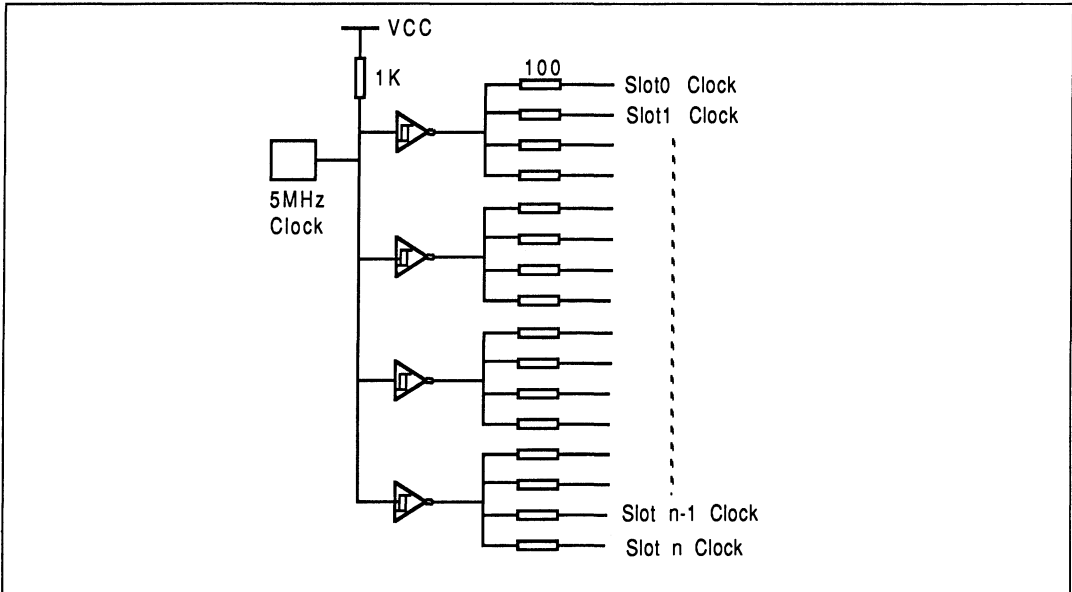


Figure 9.13 Clock distribution

9.5 Interface to a separate host

Some module motherboards may require an interface to a host machine or system that is not transputer based, e.g. the IBM PC, VMEbus or Futurebus. Because the implementation of the interface is specific to the host system, it is not defined here. However, it should allow the system to access the module pipeline and control a subsystem of modules.

9.5.1 Link interface

The host system accesses the module pipeline via Slot 0 Link 0, as shown in figure 9.14. It is beyond the scope of this document to define the implementation of the host to link interface, but it might consist of an INMOS link adapter, the registers of which may be mapped into the host's address space, or it may involve the use of dual-ported RAM shared between the host and a transputer.

The interface must be capable of interrupting the host when a data transfer in either direction has been completed.

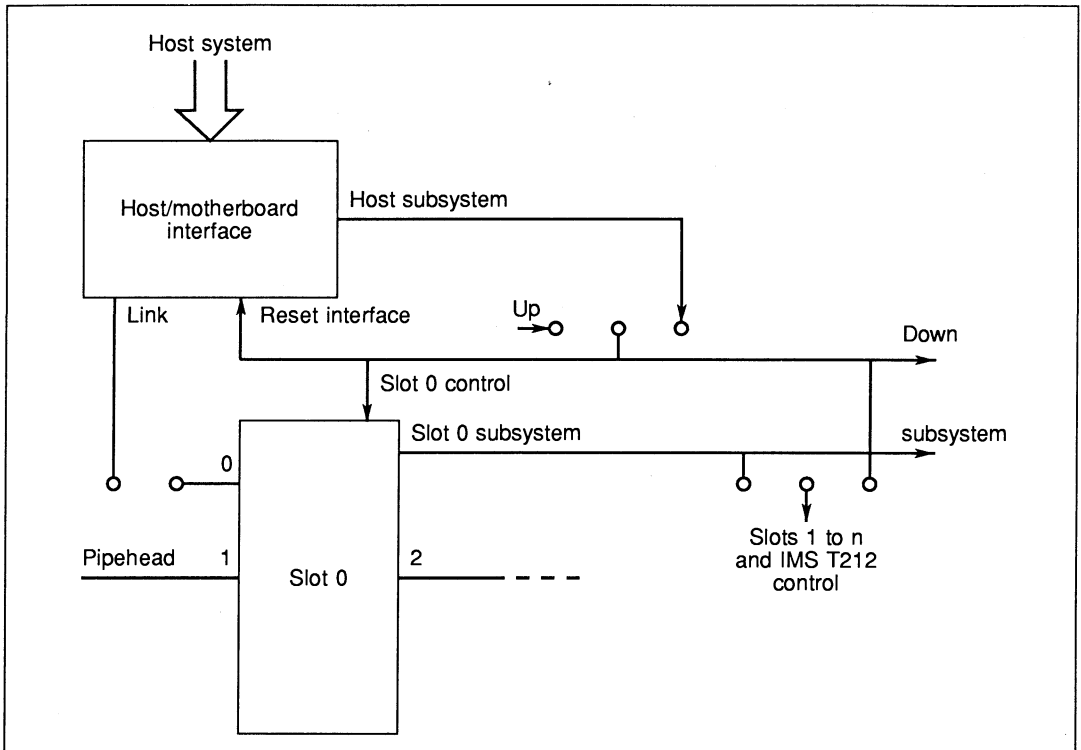


Figure 9.14 Host to motherboard interface

9.5.2 System control interface

The host system must be able to control a network of modules. This is made possible by the provision of latches mapped into the host's memory. There are three latches: Reset, Analyse and Error, which correspond to the **notHostReset**, **notHostAnalyse** and **notHostError** signals of the *HostSubSystem* port shown in figure 9.14. The Reset and Analyse latches are mapped into successive locations of host memory. Reset and Analyse are write only by the host; the Error latch is read only and shares the same address as the Reset latch.

Writing a '1' into bit 0 of the Reset latch asserts **notHostReset**;
 Writing a '0' into bit 0 of the Reset latch deasserts **notHostReset**.

Writing a '1' into bit 0 of the Analyse latch asserts **notHostAnalyse**;
 Writing a '0' into bit 0 of the Analyse latch deasserts **notHostAnalyse**.

A '1' read in bit 0 of the Error latch indicates that **notHostError** is asserted;
 A '0' read in bit 0 of the Error latch indicates that **notHostError** is deasserted.

The host to motherboard link interface is reset by the same source as Slot 0, i.e. the Up port or the HostSubSystem port.

9.5.3 Interrupts

The host to subsystem interface must be capable of generating an interrupt to the host when certain events occur on the motherboard. These include:

- Completion of transfer of data from the host to the motherboard
- Completion of transfer of data from the motherboard to the host
- Error in subsystem indicated by **notHostError** being set

Other system specific conditions may also generate an interrupt, e.g. if DMA is used to transfer data between the host and motherboard, the end of a DMA cycle may trigger an interrupt.

The host may select which conditions cause an interrupt by setting bits in a register or registers on the motherboard, mapped into the address space of the host. Other registers hold status information that can be read by the host to determine the source of an interrupt.

9.6 Mechanical considerations

The size and shape of a module motherboard is determined by its application. However, there are a number of mechanical constraints which must be adhered to in order to maintain compatibility between different modules and motherboards.

The size and spacing of module slots must conform to the mechanical specification in the *Dual-In-Line Transputer Modules (TRAMs)* document, the main points of which are reiterated here.

9.6.1 Dimensions

In the following, dimensions are quoted in inches for PCB length, width and related dimensions; all other dimensions are quoted in millimetres.

Width and length

The basic size of a TRAM is a very wide 16 pin DIP, with 3.3" between the two rows of pins. These TRAMs fit on a 3.6" pitch on their length, and a 1.1" pitch on their width. Extra length is added beyond the pins to hold the pins, to provide for mechanical fixing, and to polarise the module shape. Modules can be made larger than the standard size by keeping the 3.3" between pins and using two or more sets of the 16 pins. They can be made smaller than the standard size, down to a 16 pin DIP with 0.6" between the two rows of pins, or 1.5" between the pins. These sizes will normally be used for single chip modules or hybrids.

The top drawing in figure 9.15 shows a Size1 module and how the jigsaw pattern fits together between adjacent modules. The lower drawing in figure 9.15 shows the various sizes of TRAM. Detailed dimensions of the different sizes are given in the *Dual-In-Line Transputer Modules (TRAMs)* document.

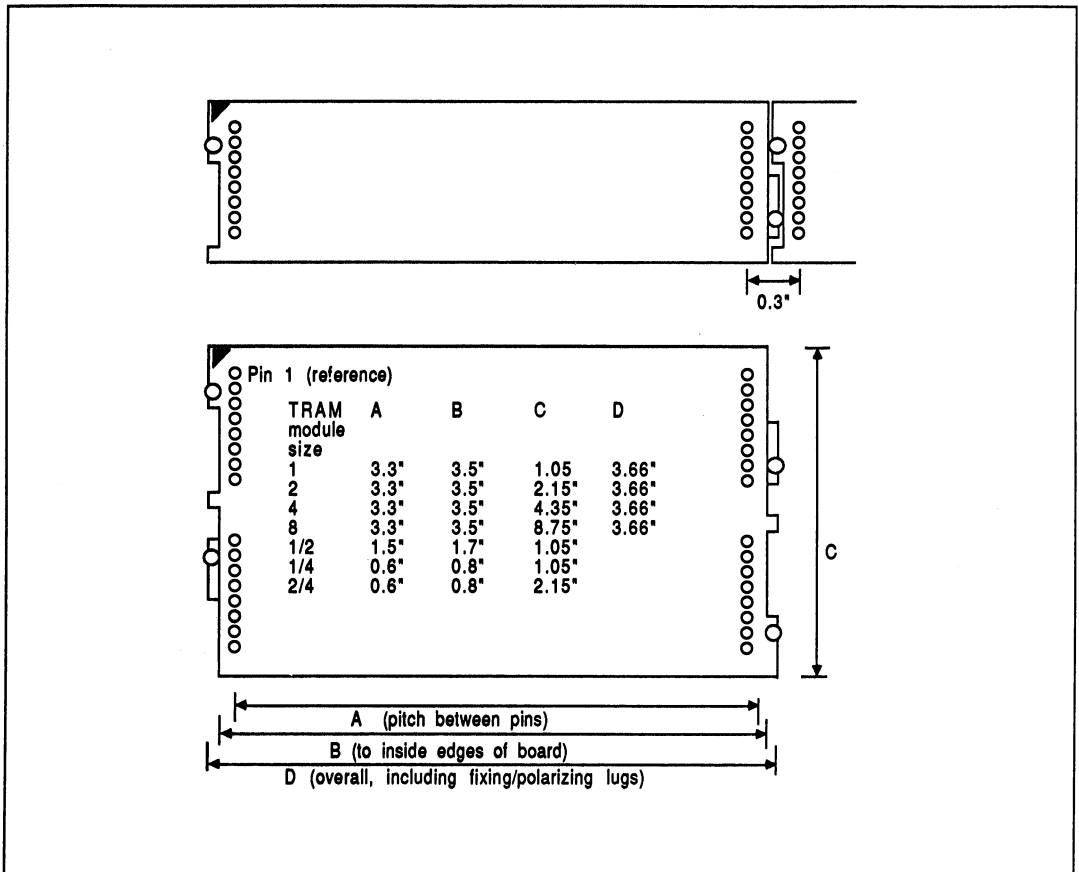


Figure 9.15 Transputer module sizes

Vertical dimensions

The height specifications, both above and below the TRAM PCB, are shown in figure 9.16a. Figure 9.16b shows a module with these dimensions plugged into a motherboard.

Figure 9.16c shows a TRAM above components on a motherboard and the overall component height is 13.7mm, which is within normal specifications for motherboards on 0.8" centres.

It is recommended that any component reaching a maximum specified height has an insulating surface.

To provide the spacing shown in figure 9.16c, the TRAM pins are implemented as a stackable socket, and an extra stackable socket is used between the motherboard socket and module pin.

Figure 9.16d shows an alternative component height which meets the 13.7mm overall height if the module is not above components on a motherboard.

Figure 9.16e shows two modules stacked.

Note that the datum for component heights on both sides of the TRAM is the component side surface. This datum is also used for the stackable socket to minimize tolerance buildup.

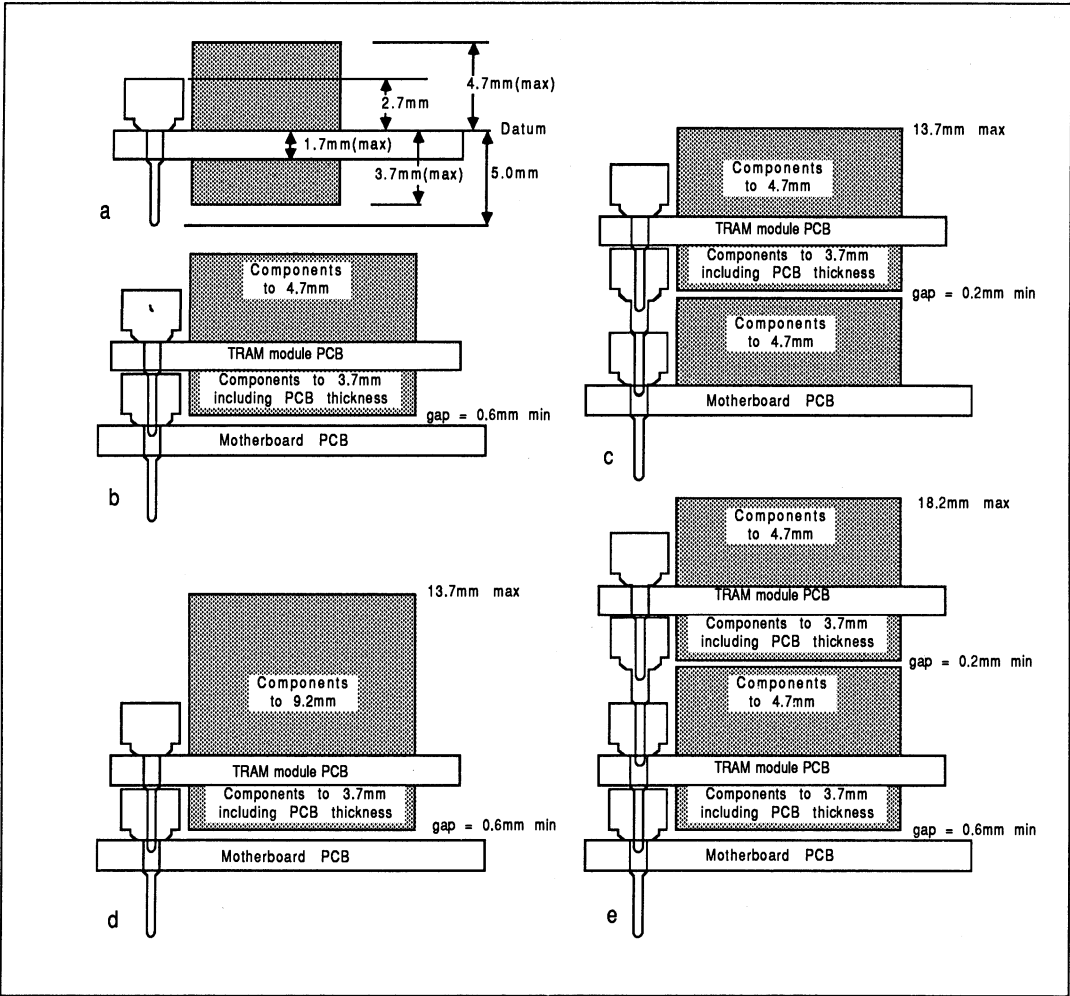


Figure 9.16 Component heights

9.6.2 Motherboard sockets

The TRAM pins/stackable sockets defined in the *Dual-In-Line Transputer Modules (TRAMs)* document will plug into any standard IC socket. To meet the component heights given in figure 9.16, the stackable socket must also be used on the motherboard.

Motherboard sockets for the Slot 0 subsystem signals should be the 0.38mm or 0.4mm sockets referred to in the *Dual-In-Line Transputer Modules (TRAMs)* document.

9.6.3 Mechanical retention of TRAMs

Vibration tests have shown that in a normal office or laboratory environment, the TRAMs remain plugged into their sockets. In transit, however, or in an environment where there is vibration, some form of mechanical retention may be necessary.

Modules have fixing holes to facilitate mechanical retention, see the *Dual-In-Line Transputer Modules (TRAMs)* document. Similar fixing holes should be drilled in the motherboard as shown in figure 9.17. M2.5 nylon bolts may be used between these fixing holes to secure the modules.

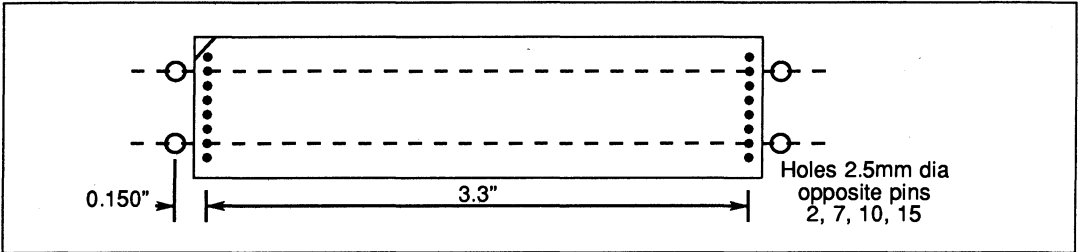


Figure 9.17 Fixing holes for mechanical retention

9.6.4 Module orientation

Figure 9.18 shows the orientation of transputer modules when mounted in slots on a motherboard. Notice how each module is rotated through 180° with respect to adjacent modules. This serves two purposes: cooling of Size 1 modules is improved; and it makes it possible to have Single-In-Line modules at some future date.

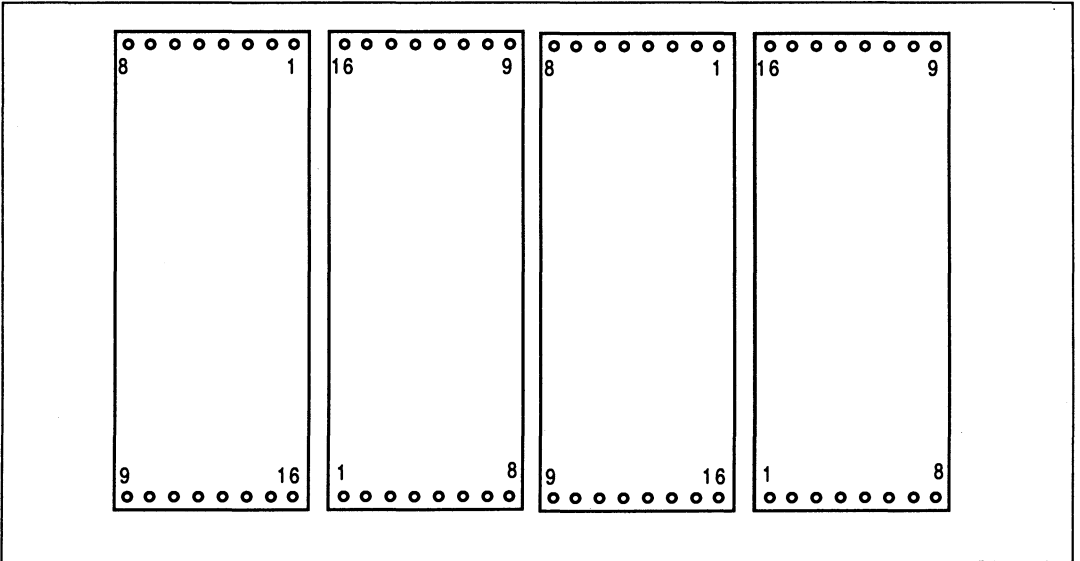


Figure 9.18 Orientation of module slots

9.7 Edge connectors

Connectors are necessary to enable links and system control signals to be taken from a motherboard to other boards. Several types of connector have been used on INMOS module motherboards.

The IMS B008 module motherboard for the IBM PC uses a 37-way D-type connector, the pin-out of which is shown in figure 9.19.

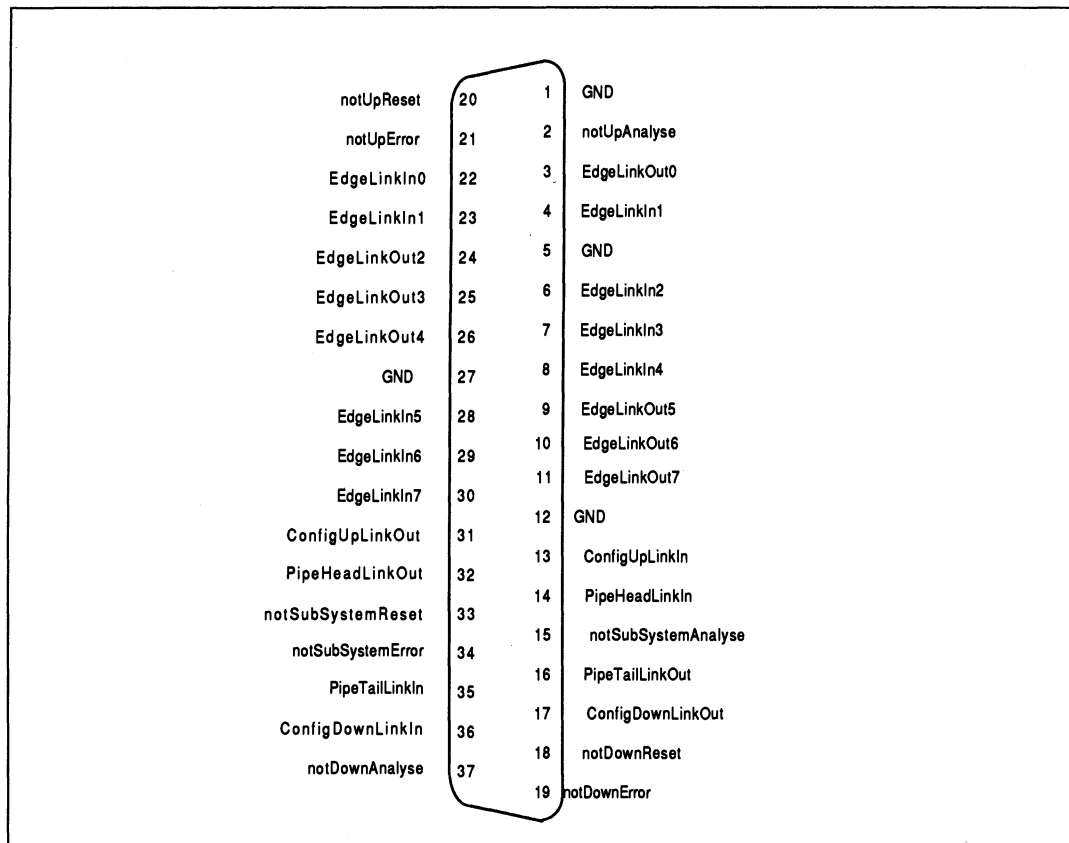


Figure 9.19 37-way D-type connector

This connector provides up to twelve links (including ConfigUp, ConfigDown, PipeHead and PipeTail), plus Up, Down and Subsystem ports. A cable suitable for connecting IMS B008s together is shown diagrammatically in figure 9.20.

The IMS B012 is a module motherboard in double extended Eurocard format. It has two 96-way DIN 41612 connectors. The bottom connector (P2) provides connections for eight links (including ConfigUp, ConfigDown, PipeHead and PipeTail) and Up, Down and SubSystem ports. Table 9.1 shows the general pinout adopted by INMOS for such a connector, making it suitable for use with module motherboards while preserving compatibility with the rest of the INMOS range of boards. The pins marked *Spare* and *Spare link* may be used for signals and links specific to a particular application. The *IMS B012 User Guide and Reference Manual* describes how these pins are used on the IMS B012.

The top connector (P1) of the IMS B012 is a DIN 41612 connector that takes a special mini-backplane to provide connections to 32 links. See figure 9.21 for the mechanical details and Table 9.2 for the pinout of this connector. On the IMS B012, the P1 connector is used to bring out links from the board's two IMS C004s. See the *IMS B012 User Guide and Reference Manual* for details. The mini-backplane is available from Varelco, part number 07-8258-0940-01-00. Both the P1 and P2 connectors are used with the INMOS Link and Reset cables provided with most INMOS board products.

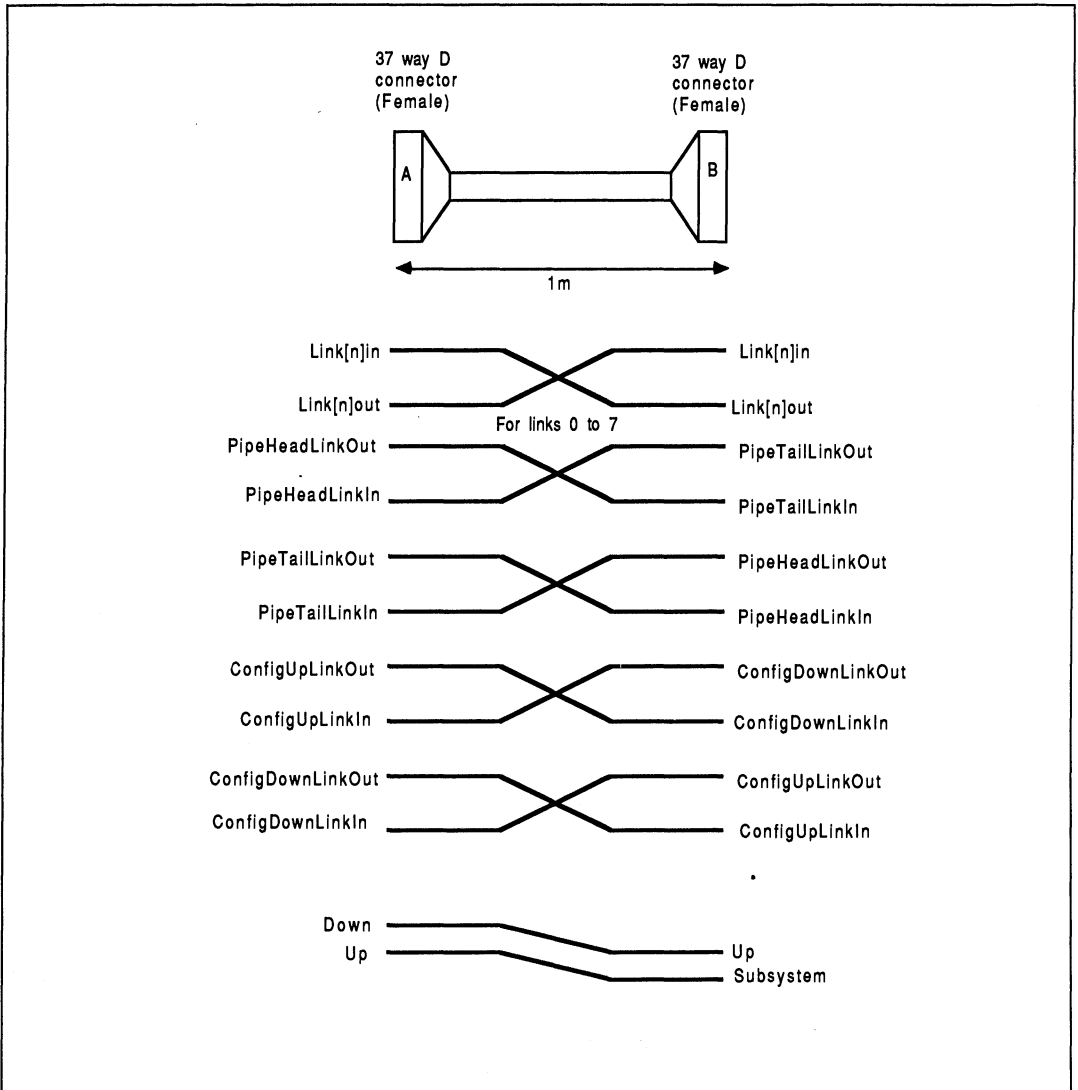


Figure 9.20 37-way cable

	c	b	a
1	GND	GND	GND
2	VCC	VCC	VCC
3	PAUX	nc	PAUX
4	VCC	VCC	VCC
5	GND	GND	GND
6	VCC	VCC	VCC
7	GND	GND	GND
8	nc	nc	nc
9	PipeHeadOut	Spare linkout	PipeTailOut
10	PipeHeadIn	Spare linkin	PipeTailIn
11	GND	GND	GND
12	nc	nc	nc
13	GND	GND	GND
14	nc	nc	nc
15	ConfigUpOut	Spare linkout	ConfigDownOut
16	ConfigUpIn	Spare linkin	ConfigDownIn
17	GND	GND	GND
18	nc	nc	nc
19	Spare	nc	Spare
20	Spare	nc	nc
21	Spare	GND	nc
22	Spare	nc	notSubReset
23	Spare	Spare linkout	notSubAnalyse
24	Spare	Spare linkin	notSubError
25	Spare	GND	GND
26	Spare	nc	nc
27	nc	GND	nc
28	notUpReset	nc	notDownReset
29	notUpAnalyse	Spare linkout	notDownAnalyse
30	notUpError	Spare linkin	notDownError
31	GND	GND	GND
32	GND	GND	GND

Table 9.1 P2 DIN 41612 connector pin out

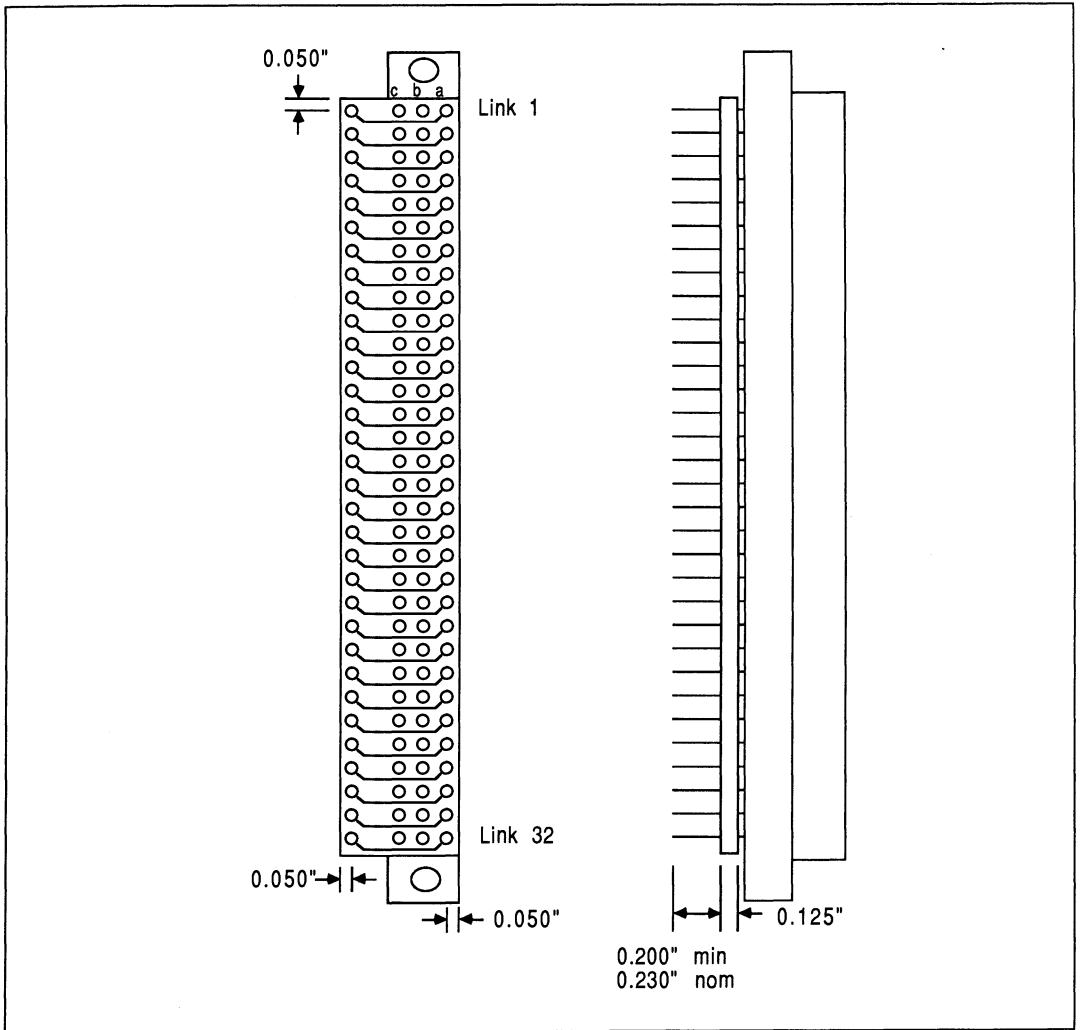


Figure 9.21 P1 32-link connector

	c	b	a
1	LinkOut0	LinkIn0	GND
2	LinkOut1	LinkIn1	GND
3	LinkOut2	LinkIn2	GND
4	LinkOut3	LinkIn3	GND
5	LinkOut4	LinkIn4	GND
6	LinkOut5	LinkIn5	GND
7	LinkOut6	LinkIn6	GND
8	LinkOut7	LinkIn7	GND
9	LinkOut8	LinkIn8	GND
10	LinkOut9	LinkIn9	GND
11	LinkOut10	LinkIn10	GND
12	LinkOut11	LinkIn11	GND
13	LinkOut12	LinkIn12	GND
14	LinkOut13	LinkIn13	GND
15	LinkOut14	LinkIn14	GND
16	LinkOut15	LinkIn15	GND
17	LinkOut16	LinkIn16	GND
18	LinkOut17	LinkIn17	GND
19	LinkOut18	LinkIn18	GND
20	LinkOut19	LinkIn19	GND
21	LinkOut20	LinkIn20	GND
22	LinkOut21	LinkIn21	GND
23	LinkOut22	LinkIn22	GND
24	LinkOut23	LinkIn23	GND
25	LinkOut24	LinkIn24	GND
26	LinkOut25	LinkIn25	GND
27	LinkOut26	LinkIn26	GND
28	LinkOut27	LinkIn27	GND
29	LinkOut28	LinkIn28	GND
30	LinkOut29	LinkIn29	GND
31	LinkOut30	LinkIn30	GND
32	LinkOut31	LinkIn31	GND

Table 9.2 P1 DIN 41612 connector pin out



Scientific language application porting and farming using transputers

10 Some issues in scientific language application porting and farming using transputers

10.1 Introduction

10.1.1 Background

Until recently, cost-effective parallel processing was not available to commerce and industry. Software was designed and implemented sequentially. Performance upgrades were achieved by using faster hardware and dirty tricks. Ultimately, though, the Von Neumann bottleneck limits the performance of such a system.

The INMOS transputer [1] avails new opportunities in performance, flexibility, and cost-effectiveness. Software can now be written to execute concurrently over a transputer network of arbitrary size, depending on the required performance.

INMOS developed a programming language called OCCAM [2] to express parallel requirements. OCCAM is the preferred programming language for the transputer. However, to protect the existing software investments of applications not written in OCCAM, INMOS provide a set of so-called scientific-language compilers for the languages C, Pascal, and FORTRAN. Ada is under development. These compilers allow applications written before the advent of the transputer to take advantage of the performance and expandability of the transputer architecture.

This document demonstrates how easy it is to use existing application software with INMOS transputers. The techniques, which are all incremental and progressively testable, do not require the application to be rewritten. Each intermediate stage produces useful operable software, allowing any amount of time and effort expended to result in an inherently better product. By observing the problems associated with porting and parallelizing existing applications, a framework and guidelines for writing future non-OCCAM applications becomes apparent.

In performing a port to transputers, there is often very little OCCAM to be written. Much of this OCCAM falls into standard frameworks, which are available from INMOS. This helps to remove some of the 'tedious' supervisory aspects.

10.1.2 Document notes

Since C developers are expected to represent the largest body of people undertaking application porting, a lot of this document will refer to C terminology and examples, but without any loss of generality. The INMOS scientific-language compilers are all handled and used the same way, as far as a mixed language application is concerned. The main software tools required to implement the techniques shown are contained within the INMOS occam-2 Toolsets.

This document does not fully explore worked solutions, but rather provides examples and offers suggestions for programmers to work with. The code fragments written in OCCAM should be readily understandable, but it is not important for the reader to understand the OCCAM in order to understand the examples and concepts.

Three dots . . . will be used to represent areas of concealed source text in both OCCAM and non-OCCAM examples. It will be assumed that any applications referred to are not written in OCCAM.

The assistance of Phil Atkin, Jamie Packer, Steve Ghee, David Shepherd, Sara Xavier, and Malcolm Boffey is gratefully acknowledged.

10.2 Preliminary information

Before discussing the porting of an application to a transputer system, there are a few preliminary details that are appropriately explained at this juncture.

10.2.1 Transputers

The INMOS transputer consists of a high-performance processor, on-chip RAM, and inter-processor links, all on a single chip of silicon. The on-chip RAM is very fast (40ns access time on the 25 MHz part), and allows fast data access and fast code execution in comparison to off-chip performance (the INMOS development tools allow the user's application to make use of the on-chip RAM [3]). The inter-processor links are autonomous DMA engines, and permit any number of transputers to be connected together in arbitrary networks, allowing extra processing power to be injected into a system very easily. The external memory interface allows linear access to a total memory space of 4 gigabytes on the 32-bit devices.

The transputer family includes 16-bit and 32-bit architecture processors. For further information on the transputer family, the reader is directed to [1]. For comparative guidelines on the most suitable transputer / board products for your application, refer to [4].

10.2.2 Processes

Transputers are hardware processors. Transputer processors execute software processes. Any number of processes can be executed on a single transputer processor at the same time. The architecture and instructions of the transputer family have been designed to efficiently support high level languages. Transputers can be programmed in conventional sequential languages such as C, Pascal, and FORTRAN.

The programming model for transputers is defined by OCCAM. OCCAM offers best support for utilizing the concurrency and communication facilities offered by transputers. Using OCCAM, a system framework can be described in terms of a collection of concurrent processes which communicate with each other and with the outside world. These processes can be written in any language. Processes not written in OCCAM can be encapsulated in a standard and simple OCCAM framework which makes them appear as an equivalent OCCAM processes (the EOP, [3]). This allows them to be used without restriction in multi-process environment.

Processes are connected together using synchronized, un-buffered, point-to-point, uni-directional communication channels. An example of this is shown in Figure 10.1, where each circle represents a process, and each arrow represents a communications channel. Each process may be written in a different language.

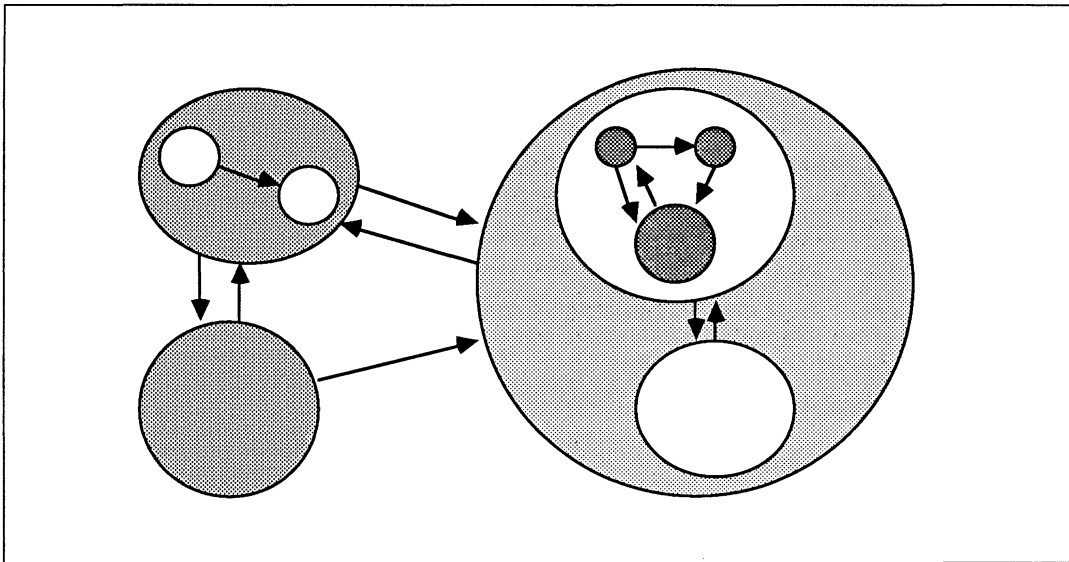


Figure 10.1 A collection of processes and their communication channels

Processes can be mapped onto transputers in an arbitrary configuration, which is independent of the processes themselves. It is not necessary to recompile any of the processes in order to change the way they

are mapped onto the available hardware.

In the context of application porting, part or all of the application will be compiled and made to appear as an OCCAM process in the system.

10.2.3 The transputer / host development relationship

In the development environment, the transputer is normally employed as an addition to an existing computer, referred to as the host. Through the host, the transputer application can receive the services of a file store, a screen, and a keyboard. Presently, the host computer can be an IBM PC or compatible, a NEC PC, a DEC MicroVAX II, or a Sun-3. One example of this arrangement is shown in Figure 10.2. In all cases, the development tools execute on the transputer connected to the host. In *addition*, the VAX- and Sun-hosted systems offer tools and utilities which execute directly on that host. For a more thorough guide to product availability, please refer to [4].

The transputer communicates with the host along a single INMOS link. A program, called a server, executes on the host at the same time as the program on the transputer network is run. All communications between the application running on the transputer and the host services (like screen, keyboard, and filing resources) take the form of messages. Software written with the INMOS OCCAM toolsets and scientific-language compilers, to use the standard INMOS servers, assume master status in a master / slave relationship between the transputer and the host. In this situation, messages are always initiated by the transputer system.

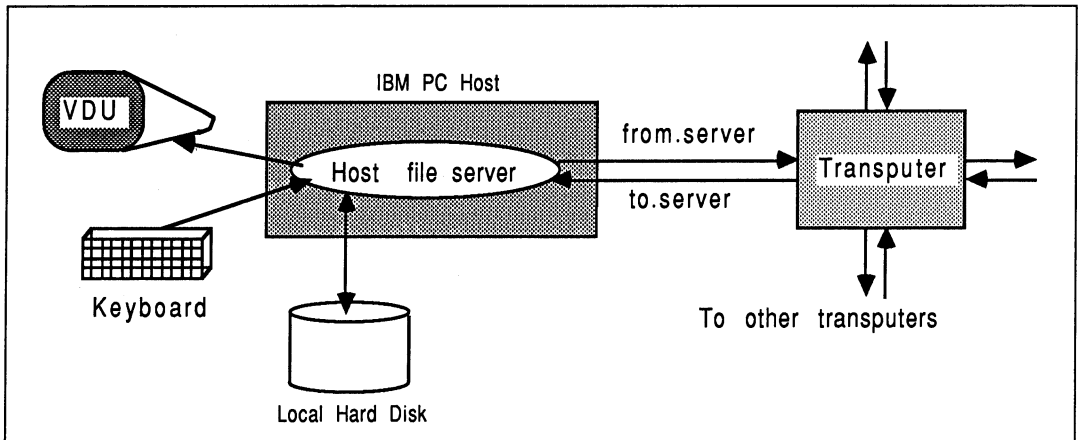


Figure 10.2 The transputer / host development relationship

The *root transputer* in a network is the transputer connecting to the host bus via the link adapter. Any other transputers in the network are connected together using INMOS links, to the root transputer. A transputer network can contain any size and mix of transputer types.

The relationship between the transputer and the host during software development does not impose restrictions on the way the transputer is employed in the target environment.

10.2.4 Why port to a transputer?

Before proceeding further, consider why one may wish to port all or part of an existing application onto a transputer system.

- **Scalable performance** : Superb performance is offered by even a single transputer. The more transputers involved, the faster the application will run.

- **Incremental expandability** : Opportunities exist for achieving greater performance through parallelizing the application (Section 10.4), and through multi-processor techniques and farming (Section 10.7). INMOS TRAMS [4] are off-the-shelf board products containing a transputer and memory. These devices can be incrementally purchased and introduced to a system as the need / means avails itself. There is no need to dispose of hardware already obtained. The software development tools permit this incremental integration without loss of development effort. Nothing is ever wasted. The hardware and software adapt to the current climate.
- **Straightforward implementation** : There can be minimal / no modification to the application source. It can be as simple as two commands to prepare an existing application for execution on a transputer. In other cases, tools exist to allow source-level manipulation of the application to take advantage of new processing power. These tools are provided by INMOS and a growing number of third-party developers.
- **Portability** : Any application executing on a transputer, using one of the standard INMOS servers, is completely host-independent. The server (thoroughly documented, of low complexity, and shipped with source and build instructions with the OCCAM toolsets) translates the host-independent commands from the transputer to the host-dependent implementation actions. Literally *any* computer with a memory-mapped link adapter and a C compiler (to build a server from INMOS sources) can be used to host transputer applications developed using the toolset development systems, because only the server must be implemented on the host. For example, transputer software written using the PC-based development tools can run *unmodified* on any supported platform, such as the Sun-3.

OK, so you just can't wait to get started. What do you hope to achieve with your port ? Let's look at a few categories of porting open to you :

10.2.5 Different categories of application porting

Depending on the requirements of the port, the time available, and the characteristics of the application, the following list outlines the categories of application porting :

1 Minimum modification porting

This involves porting all the application onto a single transputer, with no attempt at parallelization of the code. The standard services offered by the host server are assumed. This is the fastest to implement, but is the most restricted in terms of suitable applications.

2 Use with other microprocessors

For various reasons, it may not be desirable or suitable for the whole application to operate on a transputer system. In this case, a transputer system can be implemented to accommodate *part* of the application; a so-called *part port*.

3 Performance port

This involves attempting to inject some computation power exactly where it's needed in an application. The transputer software is fragmented into a small number of modules that execute concurrently, and these can then be distributed across multiple transputers using various application-specific and general-purpose techniques. Examples of this, discussed later, would be to introduce algorithmic, geometric, or farming parallelism.

Each category of porting offers a phased, progressive implementation. Each step builds on the workings of a previous, operational stage. For example, the transputer software would be initially ported without introducing parallelism, to execute on a single transputer. Then, it would be fragmented into a small number of modules, using a *stub* technique to minimize disarray in the source environment. Then, a multiplicity of transputers would be introduced. Each stage results in a useful working product, building incrementally on a working platform.

The remainder of the document discusses these categories of application porting, and the incremental tuning stages, in more detail. But first, lets survey what tools exist to help.

Transputer software development tools

INMOS provide a range of scientific-language development systems for C, Pascal, and FORTRAN. All these support the 32-bit transputer family. In conjunction with Alsys, an Ada environment is being developed, which can additionally target to the 16-bit transputers. On their own, the vanilla scientific-language development systems permit a single transputer, single process application to be constructed.

To build a multiple processor system, one is advised to use an INMOS OCCAM-2 toolset, in conjunction with the appropriate scientific-language compiler. The toolsets are available for PC, Sun-3, and VAX environments, and offer debugging facilities. The examples in this document will refer to tools for the PC environment, and in particular, the D705B OCCAM toolset. Access to [3] is useful. Remember though, that for example C software compiled using the PC development system can be integrated with other parts of an application on a VAX platform, with the ultimate intention of hosting it on a Sun-3 etc. This trans-platformal portability overcomes limitations of availability of development tools across the spectrum of platforms.

There are, of course, other development systems to select from. The INMOS Parallel C and Parallel FORTRAN systems permit multiple process systems to be accommodated, without requiring any OCCAM to "connect" the modules together.

Many third party development systems exist. In addition to the C, Pascal, and FORTRAN compilers, there are third party offerings for Lisp, Prolog, Modula-2, Strand, and BASIC. Some tools are aimed specifically at one language on one platform, offering an integrated range of compilers, profilers, and debuggers, such as Meiko's CStools for Sun-3 C applications, or Parasoft's Express. In addition, some standard libraries are available for scientific and engineering applications (such as NAG, FLOLIB etc). Caplin Cybernetics offer a range of MicroVAX development tools to allow communication between the VAX and a transputer application (the Caplin VAX/CSP libraries). A number of transputer operating systems are also available, such as HELIOS, transIDRIS, and Hobbes.

Reference to [18] will show a selection of development products available from third party developers. Make sure you get your copy. INMOS would be pleased to advise individual customers on any aspects of software development tool applicability.

10.3 Altering the application as little as possible

This section considers the simplest porting situation for an application. The application is to be lifted from an arbitrary computer system, and executed on a single transputer connected to a supported host platform.

Compliance with the goal of altering the application as little as possible requires that the entire application is executed on a single transputer. This is because the programmer can overlook the additional overheads of decomposing the application into a distributed interacting parallel system, and can use the standard INMOS OCCAM supporting software.

The goal is to modify the application as little as possible, while achieving significant performance increase.

10.3.1 The scenario

Before the porting to transputers, the application looks like Figure 10.3. No assumptions are made about the nature and capabilities of the original compute engine, except that the application uses only the following facilities through standard function calls to the language's run-time library :

- Keyboard
- Screen, and
- File system.

It is significant that the Figure does not show the application as having access to any special *host-dependent* interfacing features, and the original host's identity is not important.

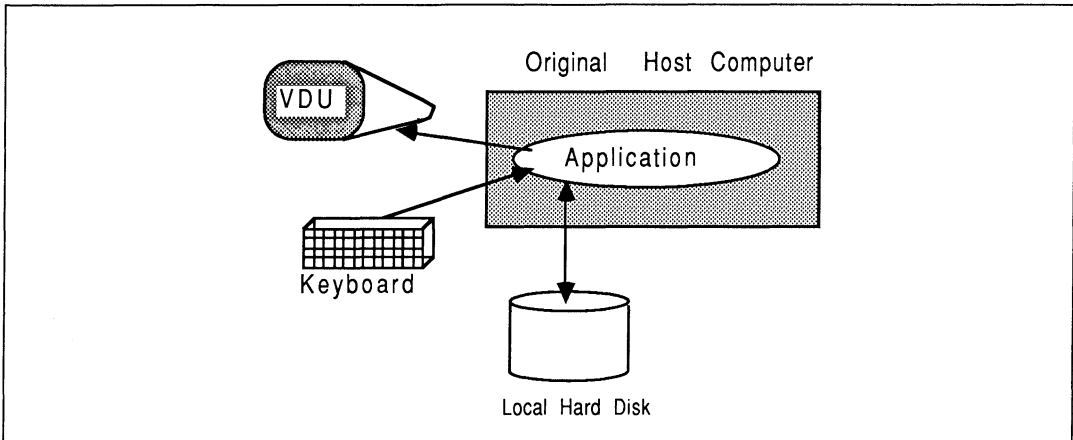


Figure 10.3 The starting point

The ported application is shown in Figure 10.4. It is shown in the context of a PC host. This can be thought of as a flat port, in the sense that no articulation or re-arrangement of the program structure is performed.

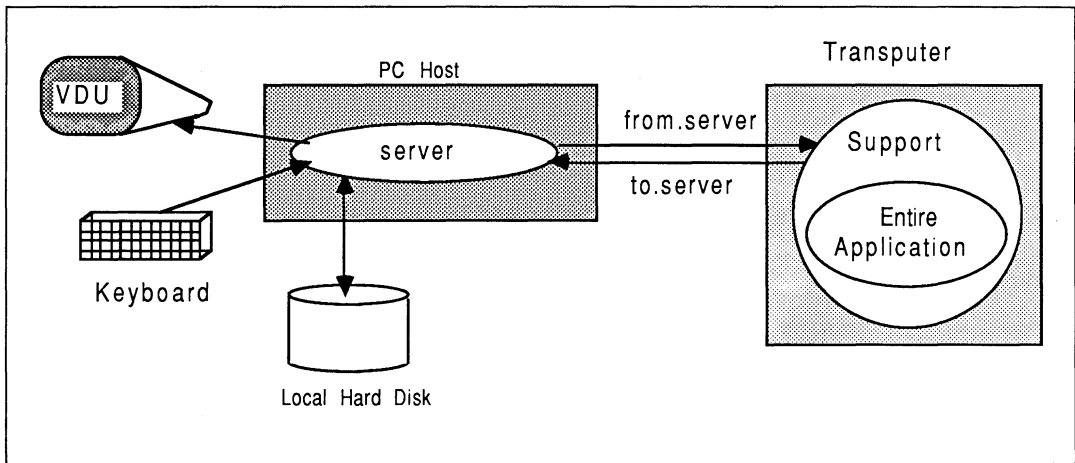


Figure 10.4 The entire application on one transputer

The host system, shown as an IBM PC, runs a simple program called a server which ensures that the access requirements of the application in terms of keyboard, screen, and filing, are fully satisfied. The standard INMOS server supplied with the scientific-language development systems and the D705A OCCAM toolset is called **AFserver**. This server is not recommended for use with application ports.

The OCCAM-2 toolsets use a different server called **iserver**. It must be stressed that all new software tools / applications should be written to use the INMOS **iserver** where-ever possible. This server is available on a growing number of host platforms (such as VAX and Sun-3) and environments (such as HELIOS). Its adoption brings immediate binary-level platform portability for the software concerned.

A small amount of software support is required on the transputer, shown in the Figure. In this specific situation of minimal modification to the application, the support is concealed from the programmer and is supplied by INMOS with the development systems.

10.3.2 Suitable applications

Requirements

For the fastest and simplest route to implement an application on transputers, then ideally all of the following apply :

- All the application source code is available, and can be easily transferred to a host platform for transputer compilation targetting. It is particularly important to remember that the application need *not* originally come from that platform.
- The application is written in C, Pascal, FORTRAN, or OCCam. Any mixture of these languages can be employed, but this involves additional steps [3] to operate the development tools. Third parties provide support for other languages, such as Lisp, Prolog, Modula-2, Strand, and even BASIC (!).
- The application source does not deviate from the appropriate International standards for these languages, to which the INMOS scientific-language compilers conform [4,3].
- The application does not use any special host dependent features like memory-mapped screen i/o, sound, etc. Keyboard input, file operations, and simple screen output, are all permissible in this category of application as long as they are performed using standard run-time library function calls. Interactive-type programs that perform simple cursor-addressing and all facilities concordant with the ANSI device driver (in the case of a PC host) can be safely placed on a transputer and used with the standard servers.

This is the scenario depicted by Figure 10.4, which also shows the server running on the host computer, and some invisible software support on the transputer.

Good candidates

Good candidates for transputer porting would be compute-intensive programs, which do not require much interaction with any host services (and therefore with a user at run-time). Generally, the smaller the amount of host interaction, the faster an application will execute. This is because availing host services usually takes a long time compared to the processing capabilities of the transputer. By avoiding user-interaction, a transputer-host combination can operate together at maximum speed without intervention. Batch-type applications are generally well suited to this, providing the quantity of file traffic is small in relation to the computation performed on the data.

Transputers can be particularly impressive with applications employing a lot of floating point mathematics.

10.3.3 Identifying the best transputer for your application

An outline of transputers has already been given. Here is a summary of some important areas in connection with device suitability in specific cases.

- If an application is integer-based, then a 32-bit IMS T414 / IMS T425 would be appropriate. These both offer 20MIPS peak performance, with the latter offering a higher performance 25MIPS version, superior link bandwidth, and additional microcoded instructions for 2-dimensional block moving (useful in graphics applications, and in block memory initialization) and CRC error detection facilities.
- If the application uses a lot of floating-point mathematics, then one should consider using an IMS T800. This is also a 32-bit 25MIPS processor, with 3MFLOPS peak performance on a 25 MHz part. The T800 can overlap floating point operations with normal CPU activity, and with link operations. A flavour of T800 called the IMS T801 is also available, which has a non-multiplexed address/data bus. This allows faster (2 cycle) interfacing to external memory, offering higher memory bandwidth.

Evaluation boards and TRAM modules [4] are available from INMOS and third parties, offering different combinations of transputers and external memory in off-the-shelf units.

10.3.4 Some potential porting difficulties

Certain parts of any application will have been fine-tuned to run on the original host computer. This tends to make parts of the application less general than they would otherwise be.

For example, the following areas can present difficulties in a total port to a transputer system :

- Some programmers use their own compilers with language extensions for their software. This could require some re-work to get the software through the INMOS scientific-language compilers. As an example, strings in Pascal can make porting difficult because almost all Pascal compilers implement this aspect slightly differently. This applies to any language extension not covered by an appropriate international standard.
- Timing facilities offered by most languages tend to be host-dependent, and fall outwith the scope of international standards. Programs making use of non-standard host-dependent facilities like this can be accommodated by making small modifications to the server.
- Some applications making *heavy* and intimate use of host-dependent hardware or peripherals are best left partly on the original host computer. For example, to *keep* graphical WIMP¹-type applications with memory-mapped screens interactive and responsive, the host-dependent graphics and mouse interfacing is best left on the host. Such an implementation is known as a part port, discussed in Section 10.6.
- Portability of tightly coded optimized assembler instructions between processors is notoriously difficult. However, most companies would write an entire application in a high level language (like C) as part of the normal application development route, and then write selected parts in assembler following program profiling. So, it is reasonable to expect that (most) low-level functions will have their high-level counterparts available for porting, in a transputer-supported scientific-language. If the high-level counterpart is not in one of the transputer's supported scientific-languages, it should be easier to express it as such than would a specific non-transputer assembler source.

If some source is not available or suitable for porting, then it must present a well-defined "interface" to other parts of the system. In this case, a small amount of glue-software could be written to mesh and interact between this code and some transputer software.

10.3.5 An implementation overview

To implement an application on a single transputer, involves three logical steps :

- **Source compilation** : All the application source must be compiled for the target transputer. The INMOS scientific-language compilation systems permit separate compilation of source units down to the function / procedure / subroutine level. This means that it is possible to take an application which is fragmented over many source files, each containing one or more functions / procedures / subroutines, and compile each file independently of any others. Once all source has been compiled, the application can be linked.
- **Object linking** : Following source compilation, the object binaries are linked together with the relevant run-time library and a proprietary OCCaM support harness. The support harness ensures that the application has correct access to the server running on the host platform.
- **Bootstrap prepending** : Before an application can be loaded onto a transputer, a bootstrap must be prepended to the linked file. The bootstrap ensures the transputer is correctly initialized before the application is loaded.

For *this* scenario, it is not necessary to make use of the INMOS OCCaM toolsets. The scientific-language development systems are sufficient for porting an application this way.

¹Windows, Icons, Mice, and Pull-down Menus

However, the **occam** toolsets additionally offer the INMOS symbolic debugger. This can be used to help identify execution difficulties in the ported application. Luckily, because in this situation the entire application is running as a single transputer process, any execution difficulties are likely to be of the traditional sequential-domain type, rather than be due to the interaction of communicating parallel processes.

The result of this implementation is an executable file for one transputer, connected to any of the **iserver** supported development platforms.

10.3.6 Porting example : SPICE

Here is an example of a real application that was ported in its entirety onto transputers, with barely any modification. An overview is presented here, but the detail can be found in [5].

About SPICE

SPICE is a large public-domain industry-standard electrical circuit simulator program. It is very computationally intensive, and is well-suited to being placed entirely on a transputer due to its total independence of the host machine. SPICE is written in FORTRAN 77, receives all its input from one file and generates all output to another file (ie, it's a batch-mode program, rather than an interactive program). It is an ideal candidate for the IMS T800 transputer due to its extensive use of floating point mathematics.

The time taken to port SPICE onto a transputer, once all the source files were available, could be measured in a number of days. There were only three files (out of 130 files) that had to be slightly adjusted to get them through the V1.1 transputer FORTRAN compiler. One part of SPICE is written in C, and used a transputer assembler-insert to establish the address of a variable. This part is the only transputer-dependent aspect.

Performance

The code compiled down to just under 500 Kbytes of object, which meant it could be run with a 2 Mbyte B004 (preferably fitted with a T800), or on a B008 with a B404 2 Mbyte T800 module. The performance figures using a 20 MHz T800 were every bit as good as a VAX 11/785, and ten times that of a Sun-3 – not bad for one T800 transputer !

The high usage of floating point mathematics in SPICE lends itself much better to the IMS T800 transputer than the IMS T414. The equivalent implementation on a T414 required almost 75k of software support for floating point routines, and the performance penalty incurred was observed to be about a factor of ten when compared to the IMS T800 on the same jobs (this is still a very respectable figure).

The table below gives an indication of the performance of some randomly selected SPICE input decks when run on a variety of different machines. Comparisons were made between a Sun-3 (with and without a 68881 numerics co-processor), a VAX 11/785 with FPA², and the IMS T800 transputer hosted by a Tandon PC.

The timings, in seconds, represent the CPU time used, apart for the T800 timings which represent the total job time including disk i/o.

Machine	Resist	Invert	Corclk	SenseAmp
Sun-3/160C	0.20	19.40	356.90	1855.50
Sun-3 + 68881	0.30	4.60	44.20	266.70
VAX 11/785 + FPA	0.38	4.51	30.22	141.55
IMS T800-20	1.48	5.17	23.72	153.04

For more information on the porting of SPICE to transputers, the interested reader is referred to [5]. This reference also discusses various farming opportunities for SPICE in more detail.

²FPA – Floating Point Accelerator

10.3.7 Porting example : T_EX

About T_EX

T_EX is a document formatting and preparation system, originally developed by Donald Knuth. Because INMOS use a T_EX macro package called LaT_EX for internal document preparation, it was decided to port T_EX to a transputer to relieve VAX CPU loading.

T_EX is most widely available in source as a large single-file Pascal program. However, a public-domain version, written in C, was obtained. It consisted of around 20 C files, each of which contained many functions. Each file was separately compiled using the V1.3 transputer C compiler — there were no difficulties involved in getting the source through the compiler. The binary objects were then linked with the standard run-time support library, and an invisible OCCAM harness. This loaded and ran successfully first time on an 8 Mbyte transputer evaluation board — the application was marginally too large for the 2 Mbyte board. Only the standard C compiler was used; there was no need to use the OCCAM toolset in this case.

A small change was made to the C source code in order to reduce the size of the boot file. This was done because the transputer C compiler handles initialization of **static** arrays by storing a byte of initialization data for each byte of the static array. In the T_EX source, there were around 5 such static arrays which were resulting in a boot file much larger than it need be. This prevented the application from executing on a 2 Mbyte board. The arrays were made non-static, and given simple run-time initialization code, resulting in a smaller boot file, which loaded and executed on a 2 Mbyte board.

For greater flexibility, a minor change was made to T_EX's path parsing mechanism to allow drive names to be specified as part of a path name. This is useful for a PC-based host, but was not necessary in the original host which would have been a sizable mini-computer. A small fragment of C was also procured to convert the time obtained from the host server using the standard C function (in seconds) into an actual date.

Performance

There is little floating point mathematics involved in T_EX. This results in a boot file around 300k in size when compiled for either a T414 or a T800. As a consequence, the performance on the T414 and T800 is very similar.

T_EX performs a lot of disk i/o in addition to heavy computation. This means that the efficiency of the server program and the link communications can have a considerable impact on the performance of the application. The table below indicates the performance achieved for different document sizes over a range of machines. The transputer is PC-hosted. The timings are given in seconds.

Machine	Loading time	1 page letter	16 page paper
PC-AT, fast disk	6.04	15.54	112.74
PC-AT with T414-20	30.32	33.40	65.98
VAX 11/785	10.96	14.37	66.82

When loading up T_EX to a transputer, a 30 second penalty is incurred while the program boots and loads up about 500k of format data. A transputer and PC combination attains a performance around that of the Sun-3, with a boot file around half as big again as that produced by the Sun GNU-C compiler. The performance degrades to around half that of the Sun-3 for larger tasks because the Sun's disk i/o is so much faster than the host PC's. However, even this is more than three times faster than PCT_EX running the same large jobs directly on an 80286-based PC. The VAX timings shown represent the CPU time; consequently the elapsed time would be much longer.

The PC version is not directly comparable because it's format file consists of 16-bit words, which makes it half the size of the 32-bit versions used by the transputer. This results in a correspondingly smaller heap requirement. The effect of this is that for *small* documents, the 16-bit PC has the advantage over the transputer implementation. But for larger documents, the transputer streaks ahead, and can also be re-run consecutively without incurring the re-load penalty.

Using the HELIOS transputer operating system [6], developed at Perihelion Software Limited, it is possible to load the transputer application and the disk-resident format files from a host PC into transputer board

"RAM-disks" before execution. This has the effect of reducing disk-bound i/o bottlenecks, and gives even higher performance.

10.3.8 Further work

Once the application port at this level of complexity is operating, a few steps can be taken without altering any of the application source to hopefully increase performance. These steps will be most effective if the application is compute-bound, rather than communication-bound with the host server.

[3] shows how to build an OCCaM harness for the application that makes better use of the transputer's on-chip RAM. This is because for applications requiring more than 512 (32-bit) words of run-time stack space, the standard harness places the whole stack off-chip, but prevents a valuable 2048 bytes of on-chip RAM being used for other purposes (like speed-critical code segments). Most reasonably-sized applications will require more than 512 words of stack space.

[3] also describes how to force the linker to order the object modules of the application to squeeze critical modules on-chip. The word module is used in this context as meaning the smallest unit of selective loading that the linker can process. A knowledge of which are the most heavily used routines in the application is required (perhaps from profiling data acquired from the original host system), but the usage of run-time library modules must also be considered.

10.4 Parallelizing the application

Following a successful implementation of an application on a single transputer, as described in the previous section, the road to increased performance begins by attempting to introduce some parallelism. This is done by expressing the application as a number of independent entities which can be made to execute concurrently. These entities will be referred to as modules, and are the atomic components of the parallelism in the system.

10.4.1 Types of parallelism

There are three broad types of parallelism that can be introduced to an application, all of which lead ultimately to multiple-transputer solutions :

- **Algorithmic** : The application is divided into modules, each one of which can execute concurrently on different transputers. This is the easiest to implement, because there is no need to change much of the existing application. The idea is to allow modules to process data concurrently with only occasional communication. There is no necessity to divide up the problem-space or modify the structure of the application.
- **Geometric** : The application input data space is divided up into independent data sets according to the "geometry" of the problem. For example, in image processing, a rectangular grid of transputers could be used to operate on an image, with each transputer responsible for a fixed area and position in the scene. This approach works best where the amount of work done in each fragment of the geometry is the same. Introducing geometric parallelism is more complex than algorithmic parallelism, because the data space must be sub-divided.
- **Farming** : Similar to geometric parallelism, the input data is partitioned into independent sets of data, which are distributed over a transputer network. The difference here is that farming is a more general-purpose technique than geometric parallelism, and the farm topology is independent of the application geometry. Farming is also best suited to applications where the unit of work varies in complexity.

It is advised to begin with an algorithmic parallelization, and then proceed to geometric or farm parallelizations as required. These classifications all utilize the same module concept.

10.4.2 Why parallelize ?

There are a number of advantages to be gained by fragmenting an application into the concurrently executing modules described in this document :

- The functional / data-flow decomposition of an application into independent and self-contained fragments allows best exploitation of the architecture and parallel processing capabilities of the INMOS transputer. Performance opportunities for concurrent processing, multiple processor solutions, farming, and pipelining are within reach.
- There is a clean, well definable interface between the module and all external modules. This facilitates development by independent teams of programmers who work only to the specification of the interface.
- The source for each module is separately compiled and linked, allowing fast implementation of specific updates and revisions to a large system.
- Any module can be re-implemented in a different way, in a different language, or on a different processor, without any impact on the rest of the system, providing it retains the same well defined interface to other modules.
- Each module's memory is logically separate from the memory used by all other modules.
- If several functions which operate on a data structure exist within a module, then since only one function in a module can be active at once, mutual exclusion of access is afforded.
- The scoping of functions is restricted to the module they are contained within. This is similar to the **PACKAGE** concept in Ada, or the **MODULE** concept in Modula-2, and has important consequences in providing security by means of data abstraction. The INMOS transputer Pascal compiler offers a similar **module** concept with good control over the visibility of functions and procedures. Instead of providing this security by a *public* specification of the procedures and functions available, one provides a specification of the channels and protocols available for use with the module.
- An implementation can be configured to run easily on a variety of hardware topologies, which also aids portability between different hosted machines. Operating systems like HELIOS [6] allow an application, comprizing of concurrent modules, to adapt at run-time to the existing hardware available. The application requests certain services such as floating-point transputers, or graphics facilities, or filestores. It is not constrained to any pre-determined hardware topology.

Converting an application to modules in an attempt to introduce algorithmic parallelism will itself only give speedup if the programmer can arrange for computation activity during periods of screen / keyboard / file-system interaction. The greatest performance benefits will come from distributing a modular system over a number of transputers.

Attempting to parallelize an application is not as simple as the flat-port to a single transputer described in the previous section, because one has to identify specific parts of the application involved that could be (profitably) executed in parallel. This is very dependent on the application concerned, and how it is structured.

10.4.3 Definitions

The following definitions apply rigorously to the remainder of this document :

- The word *function* is used to represent code units written in a non-OCCAM language. The code units could be **functions** (as in C and Pascal), **procedures** (as in Pascal), or **subroutines** (as in FORTRAN).
- A *function grouping* consists of a collection / hierarchy of functions that call each other in some structured way. This is generally the same structure (or a branch of it) as in the flat-ported implementation. The *senior function* is the single function that can be thought of as the "entry point" or "call" to the grouping.

- A “module” is a collection of function groupings which is executed concurrently with other modules. This useage of the word module should not be confused with that generally associated with separately compiled object binaries, which are not complete self-contained “sub-programs”. The module is the unit of concurrency in a system. A module represents the so-called non-OCCam process (NOP), which is made to appear as an equivalent OCCam process (EOP) by a small OCCam support harness [3].
- The *root module* is the module that communicates directly with the host server.
- Modules communicate with each other in pairs. A *super-ordinate module* “calls” or “accesses” services provided by a *sub-ordinate module*. Generally, the root module is the most super-ordinate module, ie, it ultimately controls all other modules in the system. Generally, to avoid software overheads, a sub-ordinate module is always accessed by the same super-ordinate module.
- The *module interface* is the *only* gateway for modules to communicate with each other and access data and code items. This communication always takes the form of message-passing, which conforms to a specified protocol for the modules concerned.
- The *environment* of a module represents all the data and functions contained within the module. Figure 10.5 shows each module consisting of an environment boundary, enclosing a number of items, which are code items (ie functions), and data items.
- The expressions *non-local* and *remote* are used to indicate access to code or data items in a different module. Access to code or data items in the current module are local accesses. All non-local accesses are accomplished by a communication interchange on specified communications channels between the two participating modules. All local accesses are unchanged from the original non-ported application (and the same as the flat-ported application) by reading variables or calling functions by name, for example.

Armed with these definitions, and a really hot cup of tea, consider the work involved :

10.4.4 The stages in modularizing

To take an application and decompose it into modules, the following stages will generally be involved :

- Decomposing the application into several independent function groupings that can be profitably executed in parallel.
- Enclosing these function hierarchies with a standard piece of simple source code (in their own language) to make them appear as message-passing modules.
- Generation of simple OCCam support for the modules, to make the so-called equivalent OCCam process (EOP) [3]. This allows the module to be freely used within a multiple process system.

The EOP modules, with their encapsulating OCCam, are interconnected using a simple top-level OCCam harness. Each transputer in the system will have a top-level harness to bind all the processes together. A single *configuration description* then maps all the software components onto the transputer network.

If the application is written entirely in C or FORTRAN, the INMOS Parallel C and Parallel FORTRAN development systems allow the entire interconnectivity of the application to be expressed without using OCCam. This is done using a meta-language, but achieves the capabilities as the OCCam Toolsets. This document refers to the use of the OCCam toolsets in the examples.

10.4.5 Modules

This section discusses modules, as they relate to the fragmentation of an application into a series of parallel processes. The system is under the control of a single module, derived from the main control structure of the original application. This requires the minimum of OCCam support.

Module properties

Fundamentally, the use of non-OCCAM languages on transputers is controlled by OCCAM statements to instantiate sub-programs and allocate workspaces [3]. The OCCAM model imposes certain restrictions on parallel processes in a transputer system, for example, communication is restricted to the form of messages propagating on unidirectional point-to-point unbuffered channels. Since a module is represented as a single parallel process on a transputer, modules themselves can only communicate with other modules by message passing.

A summary of module properties is now listed :

- Contains many functions from the original application, almost all of which are unaltered.
- Normally written in one language.
- Communicates with modules implemented in any language — all INMOS development systems share the same internal representations for the most common data types. Thus, characters sent by a Pascal module will be interpreted as the *same* characters by a C or FORTRAN or OCCAM module. Similarly, integers and floating point representations are standardized in so far as the languages permit them to be, allowing free interchange of standard data types throughout a mixed-language system.
- Possesses one entry point which can be used to select any of several internal function groupings.
- Communicates with other modules exclusively through channels by means of point-to-point message-passing.
- Has a completely self-contained environment, making no “background” accesses to data or capabilities in other modules except by a message-based communication.
- Independent of the hardware topology upon which it is executed,
- Is instanced very infrequently (usually only once at the start of execution of the system), but utilized very frequently by sending it work request messages and awaiting result replies.
- Terminates cleanly and in a controlled way,

These aspects are now considered in more detail.

Modules provided by the INMOS tools

The INMOS development tools offer the best support and least run-time overheads if each module performs a fairly substantial amount of work (in the computation sense).

A non-OCCAM unit in a transputer system is almost a complete sub-program. It is separately compiled. It has run-time library support linked in with it. It has a main entry point that initializes data structures. Each module has its own run-time library support, even if more than one module uses the same routines from the library. This leads to a certain overhead in memory space, which increases with the number of non-OCCAM modules in the system. In addition, there are temporal overheads involved in instancing a module, which are caused by the module start-up routines relocating static initialized data from the tail of the module code area to the run-time heap for the module.

Concerning the collection of functions contained within a module, consider Figure 10.5 :

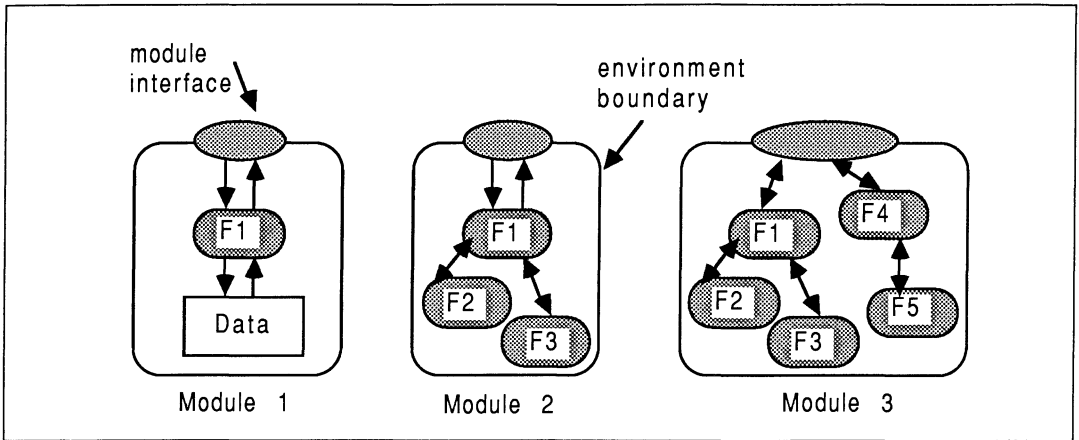


Figure 10.5 Examples of a module contents

With reference to the Figure, the small shaded rounded boxes represent functions, and the large unshaded boxes represent the boundaries of a module. Three different modules are shown :

- Module 1 consists of a single function body and some data items used only by that function. A data structure and all the functions required to access and update the data structure are generally packaged together in the same module. This is normally a sensible way to bundle a function grouping, in the sense of localizing information and its manipulation functions. As a corollary, in an implementation of this type only one function may be executing at once, thereby offering a "critical section" mutual exclusion feature of the type originally proposed by Hoare in his schema for Monitors [7].
- Module 2 shows the main entrypoint function (F1) calling other functions (within the same module).
- Module 3 consists of two function groupings, the F1, F2, and F3 grouping shown on the left, and the F4 and F5 grouping shown on the right.

A module would generally contain many functions and function groupings.

Instancing modules

Modules require some supporting OCCAM, known as the harness. The harness can be thought of at two levels. Each module is firstly enclosed in an OCCAM wrapping which allocates workspace to satisfy the stack and heap requirements of the module. This harness and the module together represent an equivalent OCCAM process (EOP). It is written as a **PROC**, allowing it to easily connect to other EOPs using only channels. Then, all the EOPs are interconnected using a top-level harness, one per-transputer. Refer to [3] for guidelines on all aspects of harnessing.

Modules communicate by passing messages. Modules can utilize an arbitrary number of channels to communicate externally. These channels are set-up by the OCCAM harness.

OCCAM statements are used to control execution of each EOP module. Because all the useful application code is confined to non-OCCAM modules, the system instantiation pattern is for the OCCAM top-level harness to instance all modules together in parallel at the start of system execution, and let the application control

itself until termination.

```
...   define interconnect channels
PAR
...   instance module 1
...   instance module 2
...   instance module 3
```

Once a module terminates, it cannot be restarted under the control of module. Only OCCAM can be used to instance a module.

Module structure

A module has to be structured such that once it is entered (upon instantiation by the OCCAM code), it does not terminate until the application has finished with it. It is useful at this point to compare the temporal existence of modules and functions.

Module communication requirements

Modules communicate with other modules using messages sent on channels. Absolutely *anything* non-local that is required by a sub-ordinate module must be copied from the super-ordinate module into the local environment. This falls into two categories :

- Parameter requirements.
- Non-parameter requirements, ie, free variables.

Some data items will only have to be read into the module, and not exported afterwards. Examples of this would be “value” parameters and free variables that are not written to in the module. Some data items must be sent both ways, for example, any “reference” parameter or written-to free-variables. Non-local data items such as strings and arrays, which are generally referred to by pointers, must have their entire body communicated (both ways if the data item could be written to) *instead* of the pointer value.

Any data item used outwith the current environment must be message passed for every interaction with the module. The messages represent the values of the parameters and free variables that would have been passed to and from the module if it were still implemented as a function call in a flat system.

Module communication protocol

It is vitally important to ensure that each interaction between two modules observes a constant, fixed, defined protocol sequence. This is the only safe way to prevent deadlock, and prevent parameters getting muddled up. For example, Figure 10.6 shows the consequences of an inconsistent message ordering in the sending and receiving environment. Referring to the Figure, if data elements C and D are the same size, then they will end up at the Controller module having each other's value. If they have different workspace requirements, then the system will possibly hang, depending on the primitives used at each end to transfer the data – but the results will definitely be incorrect.

Figure 10.7 shows a system deadlock, caused by an inconsistent ordering of channel reading, between the sender and receiver modules. Although the Worker and Controller are transferring data in the correct order, the deadlock arises from the Worker module sending data on channel 1 while the Controller is waiting on data from channel 2. This data will never arrive on channel 2 because the Worker won't send anything else until it's message on channel 1 has been accepted. If the language allowed both transfers to occur in parallel as in OCCAM then there would be no deadlock potential here.

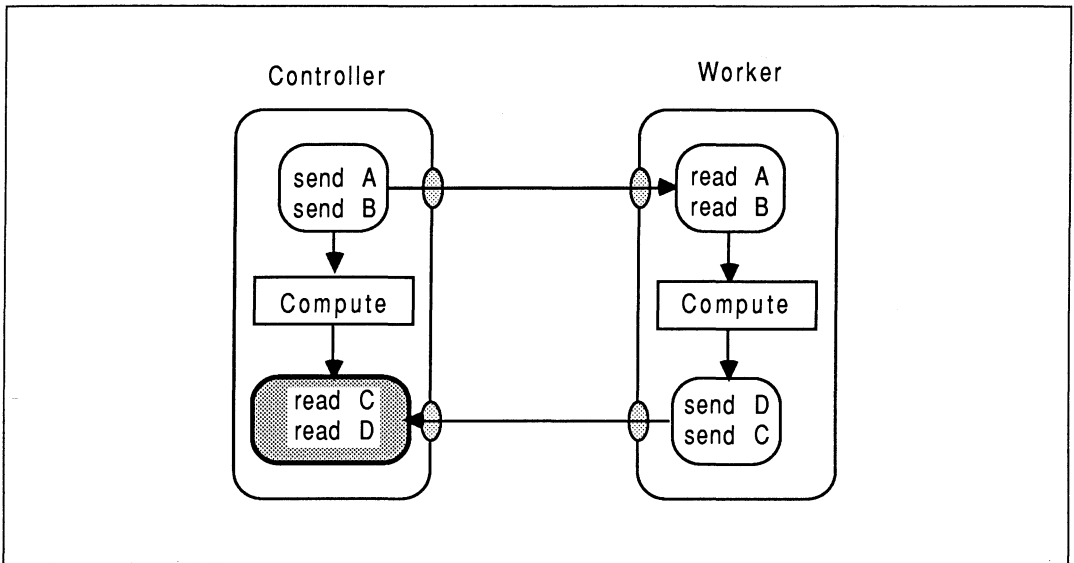


Figure 10.6 Incorrect message ordering on a single channel

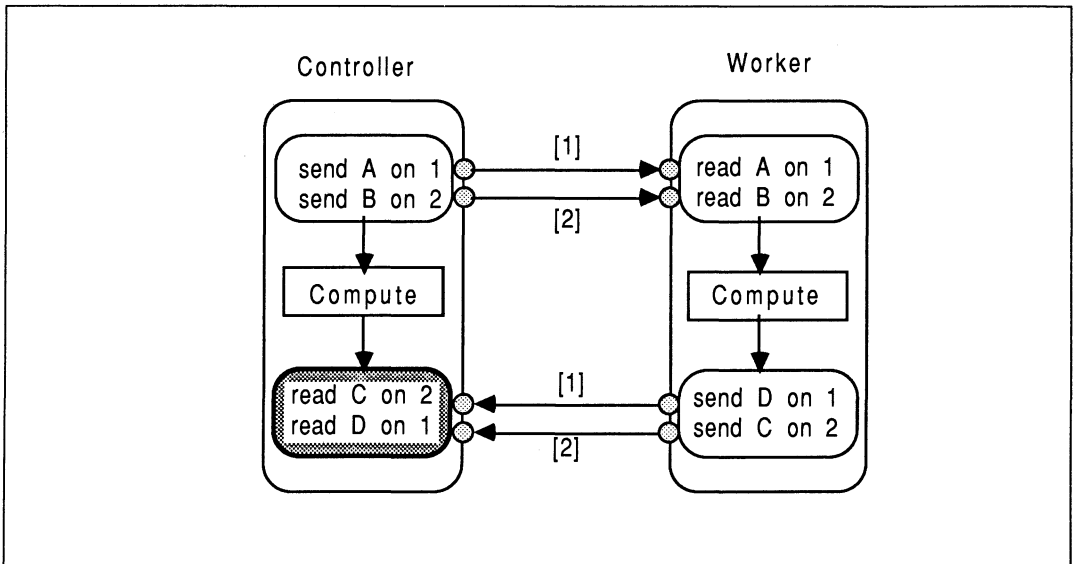


Figure 10.7 Deadlock due to incorrect channel ordering

10.4.6 Guidelines on dividing an application into modules

This section discusses some guidelines to follow when planning the decomposition of an application into independent message-passing modules. This part of an application porting is very dependent on the way the application is structured. One must have a knowledge of the data flow within the application to allow *effective*

partitioning of the program into orthogonal modules.

The objectives are that each module performs a lot of computation, but with minimal communication between neighbours.

- **Coarse Granularity** : Parallelize the application into a manageable number of separate modules, say around ten, executing concurrently on one transputer (initially). This means that within each module, there can be a large number of *completely unchanged* functions. The structure of the original application is mostly preserved, offering even better maintainability than before, and the port is correspondingly simpler and faster, and more rugged. The burden of interconnecting the modules is also lessened. In almost all cases, there are more software processes (modules) than hardware processors.
- **Minimizing inter-module communications** : Anything referenced outwith the current module's environment has to be copied into local environment store before use. One will be able to estimate the amount of inter-module traffic by the size of each item and the frequency of useage of the module by it's super-ordinate. A cost can be associated with each message interchange between two environments. Every communication adds to this cost. Aim to minimize it.
- **Restrict host-dependent constructs** : It is important to try and restrict the number of modules that make use of host-dependent services, preferably to only the root module.

For example, any construct performing screen, keyboard, or file system interaction is host-dependent. If several modules use full language input / output capabilities, then it is necessary to arrange, by means of multiplexers, for these modules to have a communication path routed to the root module that communicates with the server. Due to the limited number of links on each transputer, it is best to try and avoid the overheads of multiplexing messages along chains of modules. The best approach is to have only the root module performing full language input / output.

Supposing an application has been written in such a way that several clearly delineated operations proceed sequentially. Each operation reads some input from a file, processes it, and writes the results out to another file. A particular speech synthesis system springs to mind. If such an application were to be converted to modules with a view to distributing it across a transputer network, then there would be serious penalties paid for accessing the data files. Several approaches can be taken. One approach is to pay the disk access penalty and arrange for OCCAM software to route messages to the host for file access. Another approach is to tweak the application to write and read from a RAM-resident FIFO data buffer³. This removes the bottlenecks associated with host dependency. A further method is to use the HELIOS transputer operating system to use local processor RAM as disk storage and FIFO.

- **Profiling** : If a profiling knowledge of the application is available, perhaps from studies performed on the host system or an examination of the source code, the functions responsible for the most computationally-intensive tasks can be identified. Attempt to isolate pockets of compute-intensive activity. There is no current profiling support offered by the INMOS development systems. However, Parasoft's "Express" development system offers profiling assistance with C applications.

Note that, a detailed data-flow knowledge of the way the application is written is needed to establish blocks of code that could be completely self-contained, and operate effectively in parallel with other parts.

- **Grouping data structures and their access operations** : If one has a data structure and a number of functions that initialize and access the structure, then it could be appropriate to group the data structure and these functions into one module. The only way to access the data structure would be by these access functions contained within the module. This is most appropriate when several other super-ordinate modules would access this module, otherwise all the data and functions would be put in the same module as everything that accessed them. This object-oriented approach offers portability and freedom to alter the implementation details, providing the interfaces to external modules are unchanged.

³Note that the application itself need not be modified — an intelligent OCCAM filter can be used to intercept host-bound file access commands and convert them into FIFO accesses

The usual considerations of module traffic compared to computation performed per access should be used to establish the effectiveness of this grouping. Note that if more than one module requires to access another, a control channel and OCCAM multiplexer, shown in Section 10.5.3, is required.

- **Large or global data structures**

If a large “global” data structure exists, it is probably best to package the structure and all its access functions into a module to maintain coherency and integrity of access to the structure.

For example, a compiler or spreadsheet will contain a number of global data structures. The structures and all functions that operate on them could be packaged up as a module. This ensures consistency and integrity of the data structure and provides a clean, well-documented interface, to the outside world.

It is important to ensure that a different module does not attempt to access any parameters or free variables while they are being *used* (and possibly altered) by any other environment. This would lead to an inconsistency in the data values, which may not be appropriate behaviour for the application. This is normally prevented by a careful and strategic decomposition of the application, and by using an access protocol that forces read-modify-write operations to the data structure.

If a large collection of unrelated writable global variables have to be shared amongst a number of concurrent modules, one has to consider the overheads of broadcasting (both ways) the global data. Using protocol to “lock” access to the module (to prevent unscheduled modifications) can reduce performance because other parts of the system can become blocked. One could decide not to parallelise at the level that would require heavy overheads in frequently broadcasting and receiving global variables. Select a level of modularity that minimizes the traffic on such broadcasts, since these have to be done both ways for each access by a module.

Under very special and limited circumstances, a suitably robust application may not suffer if infrequently the “most recent” data values held in a large global array are not used for current computations. In this case, given that all the modules using this data structure are actually guaranteed to be executing on the same transputer as the data structure, then the address of the item can be used to directly write into the memory of the (single-copy) data item. This memory is *outwith* that of the module using the data item. Beware of parallel modules attempting read / modify / write operations, because there will be non-deterministic effects using this technique.

10.5 Implementing modules

Given that a ported application has been examined with a view to introducing some algorithmic parallelism, the next stage is to implement the identified function groupings as modules. A strategy for implementing the modules with minimal changes to the application is now discussed by way of examples.

10.5.1 The technique

The method involves making no changes to the bodies of any functions / procedures, or to the way in which they are normally called. It is unaffected by recursion or un-clean exiting from loops and nested conditional statements. It is independent of the topology of the transputer network. The technique is also appropriate for part-porting situations, described in Section 10.6.

Overview

Briefly, the method replaces the call to a function grouping of functions with a message-passing stub, and uses a standard fragment of non-OCCAM code in the called module to re-create the original environment of the function. This offers the immediate benefits of portability of modules throughout the hardware in a system, but without explicit parallelism between modules. Then, by a simple extension of the method, parallelism amongst the modules is introduced, offering even greater performance when used in conjunction with several transputers.

The technique is as follows :

- The group of functions to be made into a module are placed in a separate file from the functions that call them. These two files will represent two modules, which will be separately compiled and linked. Think of the modules as a sub-ordinate (containing the functions just picked out) and a super-ordinate (containing the calls/accesses to these functions).
- The communications protocol between the sub-ordinate and the super-ordinate modules, in both directions, is specified. This consists of a sequence of messages, firstly from the super-ordinate to the sub-ordinate module, then in the other direction. Remember that one channel is required for each direction. If several different functions in the module need to be individually "requested", part of the message protocol should identify which service is being requested.
- In the super-ordinate, the original definition of any functions now in the sub-ordinate module is replaced by a stub which communicates (with the sub-ordinate) using the agreed message protocol. This ensures that *all* references to a service in the sub-ordinate are converted to a message interchange. The stub takes the same name as the function(s) involved.
- In the sub-ordinate, a standard format in the same language is wrapped around the calls to the functions in the module. This standard format will communicate (with the super-ordinate) using the agreed message protocol, and also ensure that the module does not terminate until requested specifically.
- The OCCAM support for each module is written according to guidelines in [3]. The support ensures each module has sufficient workspace. The modules are connected to each other using standard OCCAM channel specifications.

Consider now the benefits provided by the technique.

Benefits

The technique is a fast and safe way to implement parts of an application in parallel, because :

- There is no change to either the function bodies or their parameters.
- In the super-ordinate (calling) environment, only one point in the code has to be changed to call a module, using message passing. Stubs detach all references to a module from its physical position.
- All references to the function within the calling environment are automatically intercepted by the stub, and fired off to the module performing the behaviour of the original function.
- The stub conveniently collects together the values of any global variables and strings etc required by the module, and can re-assign to these "globals" after the module returns control.
- It accommodates assignment of function values from a function **return (data)** operation.
- It allows simple alteration of the module access protocol in the calling and receiving environments, because all message passing is localized into just one function in each case.
- New capabilities can be added to a module by virtue of a tagged protocol used to identify the service requested.

Figure 10.9 shows how stubs impact the execution of two modules, M0 and M1. The shaded boxes represent active processing. Module M0 processes activities 0, 1, 2, 6, 7, 9, and 11. Module M1 processes activities 3, 4, 5, 8, and 10. Notice that the system about to be described does not yet allow any explicit overlapping of module processing.

10.5.2 Example of module implementation

It is assumed that the application has been decomposed into several groups of function hierarchies, as shown in Figure 10.8. The lines represent hierarchy between the functions, with an implied reference to some free variables (global data).

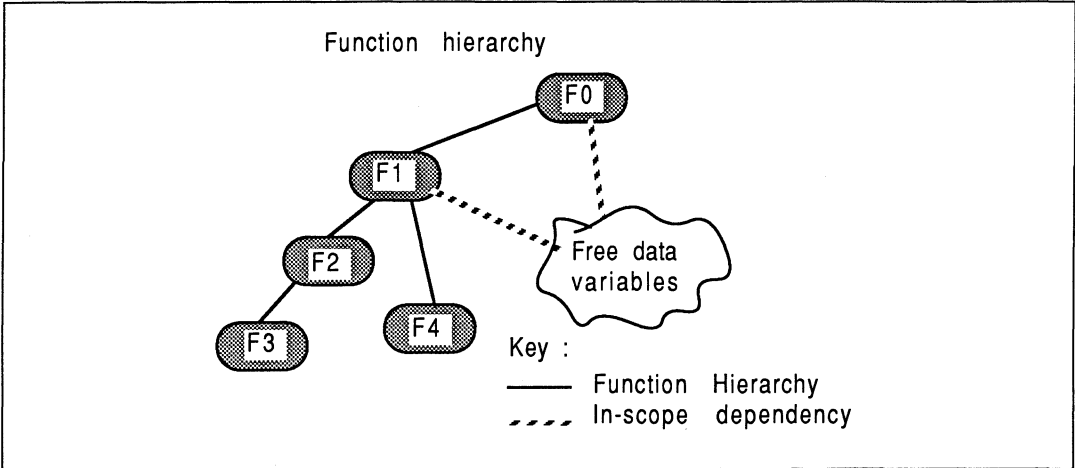


Figure 10.8 Functions to be modularized

Function F1 is considered to be the top-level function in the grouping comprising F1, F2, F3, and F4. F1 is itself called from F0, which will be represented by a different module. The method of converting the F1 grouping into a module, and referencing it from the F0 module, will now be discussed, with the objective of not changing the content or declarations of any of the functions / procedures within the F1 grouping, and also not changing any actual "calls" to the F1 grouping made within F0. These objectives are realized using the technique described above.

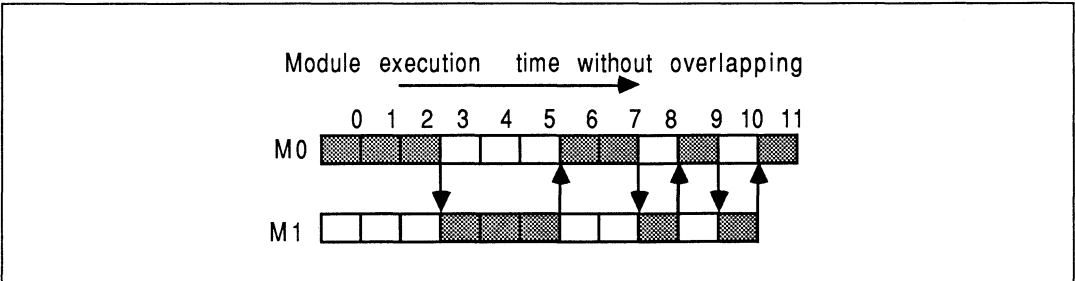


Figure 10.9 Module execution pattern using stubs

The creation of a sub-ordinate module (M1) and its reference from a super-ordinate (M0) are explained. An overview of what the two module system will become is given in Figure 10.10.

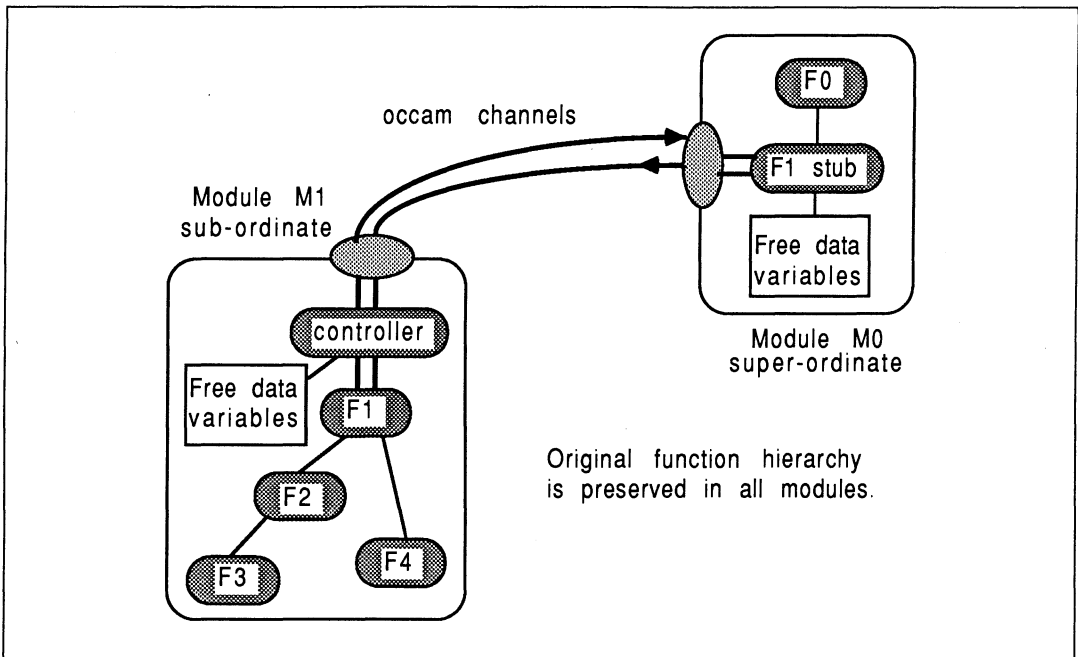


Figure 10.10 The twin module system

- Before changing anything

In the pre-modular system, suppose the C function F1() looks something like this :

```
int F1(param1, param2, param3, param4)
int param1, *param2;
double *param3;
char param4[];
{
    ... local variables defined and initialized
    ... COMPUTE !! (Calls to F2, F3, and F4)
    return(answer)
}
```

F1 is the top-level function in this grouping. It references functions F2, F3, and F4.

- Making the M1 sub-ordinate module

The M1 module consists of F1, F2, F3, and F4. The call to F1 is enclosed by a controller loop, to ensure the module never terminates until specifically instructed. This is represented by the controller in Figure 10.10. Because the functions are written in C, the standard structure for M1 is also written in C :

```
main(argc, argv, envp, in, inlen, out, outlen)
int argc, inlen, outlen;
char *argv[], *envp[];
CHAN *in[], *out[];
{
    int running = 1, tag;
    ... decls for non-local data

    while (running)      /** main loop **/
    {
        ... receive tag from calling process
        if (tag == COMPUTE)
        {
            ... receive and unpack data from super-o
            F1(value, &reference, &bigref, string)
            ... pack and return data to super-o
        }
        else if (tag == INITIALIZE)
            ... receive and unpack data from super-o
        else running = 0;    /** termination **/
    }

    ... source for F1 and dependent functions
}
```

Using this arrangement for calling F1 means that *none* of the bodies or parameters for any of the constituent functions in the module need be changed. All parameters needed are received and unpacked by the surrounding controller loop. All non-parameter variables required are also received as messages and made available in the background, exactly as in the original environment. All results and changed free variables are packaged and returned as messages to the stub that "called" the module. Don't worry about the arguments to `main()` — they are required to allow the message communications [3].

In effect, the structure described above creates the *exact* environment that the function would have experienced in its original place.

- **Making the M0 super-ordinate module reference the M1 module**

The M0 super-ordinate module contains a stub function F1 which takes the same parameters as the original F1. It's purpose is to perform message-based communication with the M1 module. The structure of the F1 stub could look like this :

```
int F1(param1, param2, param3, param4)
int param1, *param2;
double *param3;
char param4[];
{
    ... send messages to M1 module

    ... receive messages from M1 module

    return(answer)
}
```

The stub contains *only* the message passing aspects required by the real body of F1 in the other module. Figure 10.10 shows the stub of F1 being called from F0. The actual computation, originally performed by F1, is now performed by module M1, also shown in Figure 10.10.

This technique can be applied to software being used with any of the INMOS development systems.

10.5.3 Implementation notes

There are a few general implementation points to note here :

- **Start at the bottom**

It is important to convert the most subordinate items into modules first. This is so that when a super-ordinate item refers to functions within a sub-ordinate module, all the sub-ordinate's access channels and protocols will have already been defined — all the information required to reference the other module is available.

- **Protocol**

It is advisable to use the equivalent of an OCCAM tagged protocol when defining the protocol for access to a module. As well as identifying which one of several possible senior functions are to be executed, it is easily extended to incorporate new facilities. For example, module termination is cleanly addressed using this technique, simply by the use of an additional tag. It also simplifies including a module loop-back "debugging" mode which can be used to test that messages are being sent, modified, and returned correctly.

In this example, the tag **INITIALIZE** is used to handle an infrequent distribution of system data which does not form part of the regular interchanges with the module (to minimize traffic). Here is

an outline for the controller loop in a module. Notice how simple it is to implement a termination tag.

```

main( ... )
{
    int running = 1, tag;

    while (running)
    {
        ... receive tag from super-o module
        switch (tag)
        {
            case COMPUTE:
                ... handle computation request
                break;
            case INITIALIZE:
                ... handle initialization request
                break;
            case NEWMODE5:
                ... handle this computation request
                break;
            case DEBUG:
                ... handle debug mode request
                break;
            case TERMINATE:
                ... handle termination
                /** sets running = 0 **/
                /** may propagate terminate signal */
                break;
        } /* switch */
    } /* while */
}

```

The **COMPUTE** tag could indicate normal work for the process, and would invoke a standard message interchange between the calling module and called module. Imported free variables would be declared outside of `main()`, and assigned to as part of the message input protocol. The **DEBUG** tag could be used to select a different operation mode tuned to debugging, perhaps to produce additional message traffic or return checkable results. A retro-fitted function grouping could be accessed with **NEWMODE5**.

• System termination

It is important that *all* processes in a transputer system complete their application processing cleanly. This is often done by the root module initiating a termination condition which spreads to each module in the system on a predetermined route. This option is pursued in Figure 10.11, where the termination signal propagates clockwise from the root module. This can be implemented using a termination tag described above, which is forwarded to sub-ordinate modules from super-ordinates.

The shut-down is more secure if each module *handshakes* the shut-down with all its sub-ordinates before handshaking with its own super-ordinate. Modules are therefore shut down remotest first.

It is possible to arrange for the root module to send a terminate command to the host server, and neglect to shut down any of the modules. This causes control to be returned to the host operating system, but the transputer network is left running. This can allow the system to be re-run without re-booting the transputers.

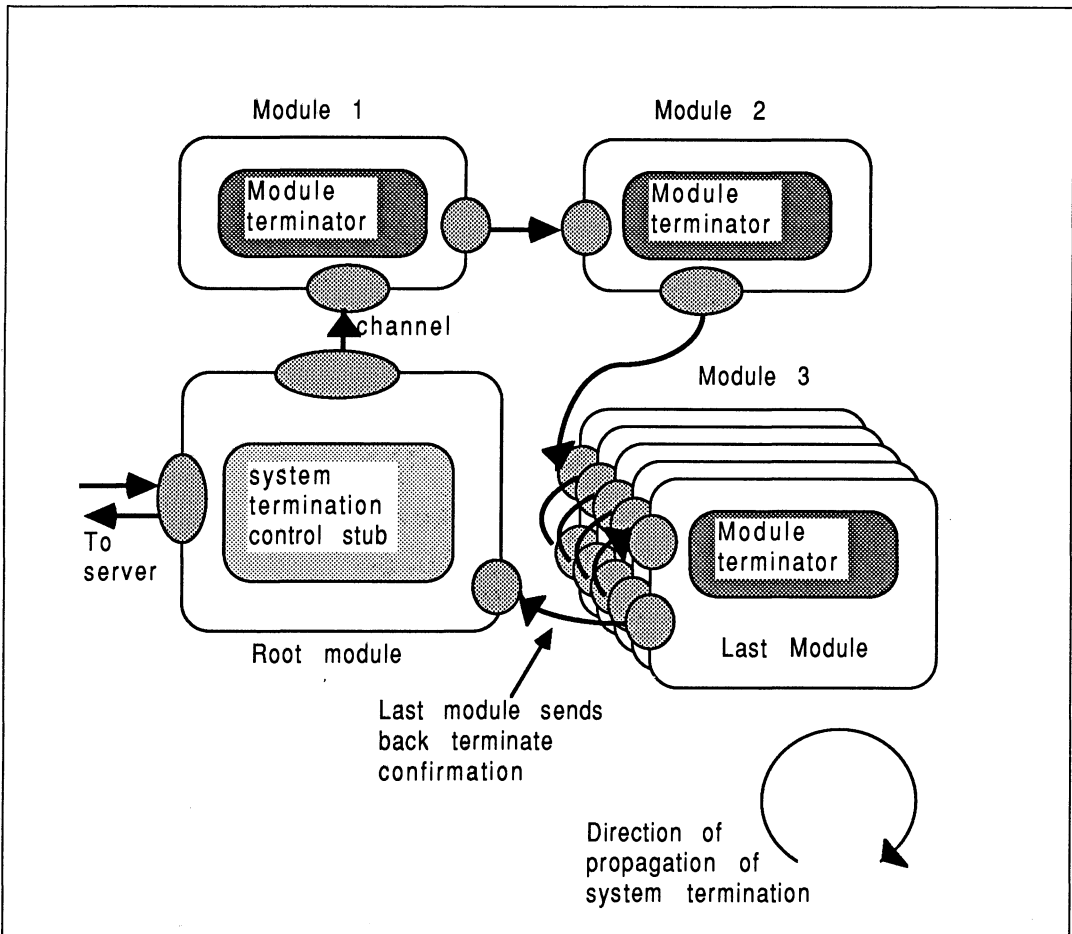


Figure 10.11 A system termination example

- **Why there's no name clashing**

Once F1 has been implemented using a stub, it means that within the system, as a whole, there are two entities called F1. One is the *original*, real, F1, and the other is the stub used to call it. This does not lead to any name-clash problems, because each module is separately compiled and linked before inclusion to the rest of the system.

In general, names of stubs, real functions, or data item do not clash with those in other modules.

- **Multiple super-ordinates**

If a module has to be accessed from many super-ordinate modules, then it is important to realize that *all* the tag messages will be sent on the *same* channel. This is because most non-OCCAM languages do not permit the simultaneous testing of data arrival on several channels (akin to the OCCAM ALT). A simple OCCAM ALT would be used to funnel access request tags from all super-ordinates to the module. A module expecting access from several super-ordinates has, as part of the access protocol, an identification tag which it uses to select channels for communication with the successful super-ordinate. The actual data messages (ie, everything except the tag) travel on

different channels, an input and an output channel per super-ordinate. Only the identity of the current super-ordinate must be sent on the same channel, which is why a simple OCCAM **ALT** is used to select one of several.

This is shown in Figure 10.12.

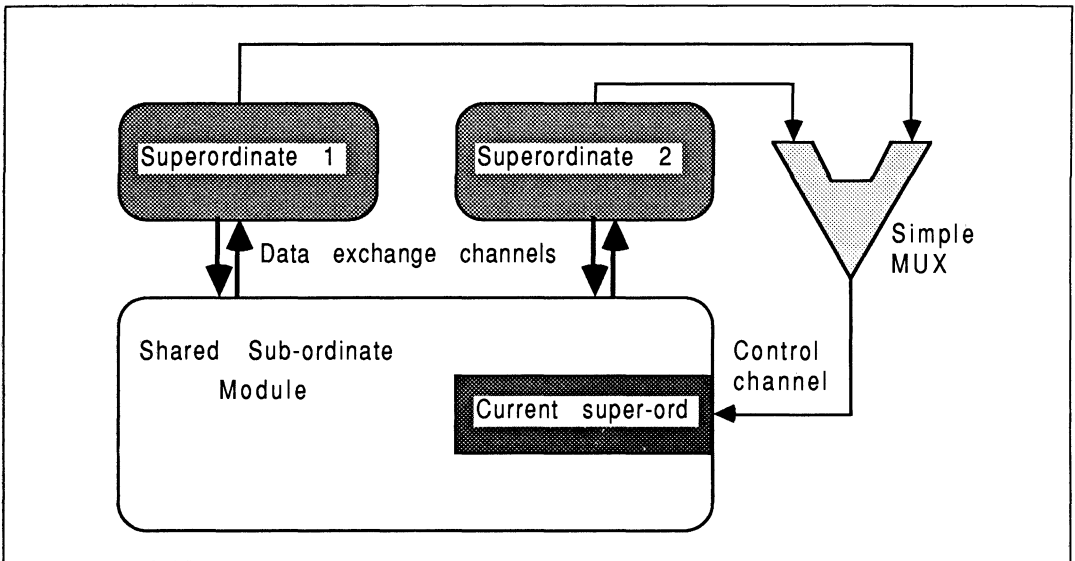


Figure 10.12 Sharing a common sub-ordinate module

To terminate the multiplexer, *one* of the super-ordinates should have an additional channel, and as part of its termination protocol it terminates the multiplexer.

- **Handling compile-time shared data**

If a large global *compile-time constant* base exists with respect to a number of modules, that has to be available to a system of processes, then the easiest way to handle this is to place definitions shared between several modules in an **#include** file (for C and Pascal). With C, the pre-processor **#define** facility would be used to prevent declaring run-time storage for any of these constants.

In this way, changes to the values of any compile-time constants can be easily made available to all separately compiled modules that need them. Unfortunately, the V1.1 FORTRAN compiler does not support file inclusion mechanisms because this is not part of the ANSI X3.9-1978 standard⁴, so one would have to make explicit textual inserts for all global constants (which would probably take the form of **COMMON** and **PARAMETER** statements).

Since compile-time constants, such as C's **#define** do not occupy any storage, there is no penalty in having them **#included** in lots of modules.

Note that with the current INMOS C compilers (V1.3 and 2.00), **static** initialized arrays are implemented inefficiently with respect to the size of boot file. A static initialized image is stored with the bootable code, meaning that large arrays lead to large boot files. Try to avoid using static initialized arrays, especially those defined across several modules.

- **Handling run-time shared data**

To ensure the integrity and consistency of run-time initialized data shared between several modules, with each module performing the initialization rather than a broadcast reception, any source code

⁴Version 2.00 Parallel FORTRAN does support file inclusion

shared by more than one module should be `#included`. This ensures that all modules supposed to hold the same data actually *do*.

There are cases where the infrequent broadcasting of global constants, which are initialized at run-time by another module, is very useful. For example, where a module has to perform some intensive computations to initialize a data structure which is thereafter unaltered but shared between several other modules. Another example might be where a shared read-only database has to be loaded once from slow disk drives and is broadcast once to modules requiring it.

In these situations, this information can be broadcast once only to all the modules that require it, allowing smaller message protocols to be used for all subsequent interactions.

It is straight-forward to implement any number of these data broadcasts, using the tagged message protocol described. The sizes of these broadcasts are not so important because of their infrequency of occurrence (usually only once), and also because the transputer can overlap computation with message passing. Each module in the system may require a different set of global constants to be sent to it, because the concept of globality is made with respect to the given module.

If the initialization of the run-time shared constants is trivial in the computation sense, then instead of calculating once and broadcasting, each module can calculate locally as well as store locally. If too many modules require *frequent* access to shared run-time constants, then the application may be better decomposed.

10.5.4 Some coding examples

This section contains some coding examples of handling parameters and accessing non-local environment data, with stubs and the message-passing functions. The examples are trivial, in that they only show the handling of a single type of parameter or free variable at once. However, by combining the techniques shown, the interface to a module of any complexity can be derived.

Consider again the F1 function grouping in module M1, being called from function F0 in Module M0. These examples assume that communications between the modules take place using element 2 of the input and output channel vectors of each module. So, the C examples use `out[2]` and `in[2]` for communication, using messaging functions called `_inmess` and `_outmess` from the C run-time library. Although not shown, each sub-ordinate message fragment is within the main controller loop to ensure services are available until the module is instructed to terminate.

- **Scalar value parameters**

A value parameter is one for which any changes that may occur to it in a function / procedure, are not reflected back to the caller. In C, consider a real value parameter called `by_value` sent by the stub F1 in module M0 as follows :

```
int F1 (by_value)          /** stub **/
float by_value;
{
    _outword(COMPUTE , out[2]);
    _outword(by_value, out[2]);
}
```

This would be received by the co-ordinator in sub-ordinate module M1 as follows :

```
float by_value;          /** local storage **/
{
    _inmess(in[2], &by_value, 4);
    F1(by_value);
}
... Body of F1 in here
```

This is the mechanism used for all scalar value parameters. Consider the same situation in Pascal.

Here is the stub, placed in the calling module :

```

procedure F1 (ByValue : real);    {stub}
begin
    outmess(channel, COMPUTE,4);
    outmess(channel, ByValue,4);
end;

```

This would be received by the co-ordinator in sub-ordinate module M1 as follows :

```

... Body of F1 in here
var ByValue : real;            {local storage}
begin {main body}

    inmess(channel, ByValue,4);
    F1(ByValue);

end.

```

• Scalar reference parameters

A reference parameter is one for which any changes that may occur to it in a function / procedure, are propagated back to the caller. In C, this is implemented by passing in the address of the item to be used, allowing changes to be directly written into that item. With modules, the actual data (and not just the reference to it) must be passed. Here, a reference parameter called **by_ref** is sent by the stub F1 in module M0 as follows :

```

int F1 (by_ref)                /** stub **/
int *by_ref;
{
    _outword(COMPUTE, out[2]);
    _outword(*by_ref, out[2]); /* sent data */

    _inmess(in[2], by_ref, 4); /* received data */
}

```

Notice that the new value for the changed parameter is slotted back into the same memory location as original — the stub does not declare additional storage for it.

The corresponding communications in the co-ordinator in sub-ordinate module M1 are as follows :

```

main( ... )
{
    int by_ref;                /** local storage **/

    _inmess(in[2], &by_ref, 4);

    F1(&by_ref);              /** call F1 the same way! **/

    _outmess(out[2], &by_ref, 4);
}
... Body of F1 in here

```

Parameters that are not four bytes long are handled exactly the same way as four byte parameters, except that the predefines **_inmess** and **_outmess** are used. All parameters can be handled this way, even complex records and structures.

In Pascal, the situation is very similar to that in the previous section, except that the Pascal keyword **var**, used to define reference parameters, indicates that the value must form part of the outgoing message protocol as well as the incoming message protocol.

With FORTRAN, *all* parameters are passed by reference. Examination of the code would indicate whether the parameter must be returned to the calling environment, or whether it can never be changed.

- Function value returns

Supposing F1 happened to return a function value, which may have been used in the original environment like this :

```
answer = F1 (&by_ref);
```

This is easily implemented using the stub approach. An extra message is used in the communications protocol for the result. The stub in the super-ordinate becomes :

```
int F1 (by_ref)          /** stub **/
int *by_ref;
{
    int result;          /** F1's return value **/
    _outword(COMPUTE, out[2]);
    _outword(*by_ref, out[2]); /* sent data */

    _inmess(in[2], by_ref, 4); /* received data */
    _inmess(in[2], &result, 4);
    return(result);
}
```

Here, the stub declares local storage for the return parameter.

The corresponding communications in the co-ordinator in sub-ordinate module M1 are as follows :

```
main( ... )
{
    int by_ref, answer;    /** local storage **/

    _inmess(in[2], &by_ref, 4);

    answer = F1(&by_ref); /** call F1 the same way! **/

    _outmess(out[2], &by_ref, 4);
    _outmess(out[2], &answer, 4);
}
... Body of F1 in here
```

- Strings

Variable length messages, for example, strings, must be handled by sending as part of the protocol the length of the string to send. By specifying the length of the string before the actual byte vector containing the data, source and destination modules always know how much data to expect.

A particular efficiency observation is appropriate for C programs. Rather than use the C function `strlen()` to calculate the current size of the string, it is faster to block-send the entire area reserved for the string. As well as avoiding timely computation in determining the string size, the block transfer can be overlapped with useful computation in other modules. This is true even for strings occupying only a small part of their reserved storage area.

In C, the zero byte ('\0') is used to denote the end-of-string sentinel. In OCCAM and other languages, this is not necessarily the case. Therefore, a little recipient-end processing is useful. Any C recipient module receiving from a non-C module must append the zero byte sentinel before availing the string to any other C routines. The position can be determined from the length information prepended to the communication.

As an example of handling strings, consider a C source and C destination module. The constant `MAXSTRINGSIZE` is declared in both modules at compile-time. Here is the code for the stub in the

source module :

```
int F1 (string)      /** stub **/
char string[];
{
    _outword(COMPUTE, out[2]);

    _outmess(out[2], string, MAXSTRINGSIZE);

    _inmess (in[2] , string, MAXSTRINGSIZE);
}
```

The C destination sub-ordinate contains the following :

```
main( ... )
{
    char newstring[MAXSTRINGSIZE];    /** local storage **/
    int len;

    _inmess(in[2], newstring, len);

    F1 (newstring);

    _outmess(out[2], newstring, len);
}
... F1 body in here
```

If the source module were not implemented in C, the zero byte sentinel should be appended in the C destination. This would also require the transmission of the true length of the string, rather than the maximum possible length.

• Pointers

Similarly to strings, any items that are referenced by pointers, either through parameters or free variables, must be sent in their entirety. If any alteration *could* be made, the entire item must be passed back to the calling process after use. As an example, consider a small array of double-length floating point numbers required by a C module environment. Given that the number of elements in the array is declared as a **#define** compile-time constant called **SIZE_VEC**, then the stub in the calling environment might look like this :

```
int F1 (dbl_array)  /** stub **/
double dbl_array[];
{
    int i;
    _outword(COMPUTE, out[2]);
    for (i=0; i<SIZE_VEC; i++)
        _outmess(out[2], dbl_array[i], 8);

    for (i=0; i<SIZE_VEC; i++)
        _inmess(out[2], dbl_array[i], 8);
}
```

Each element of the array occupies 8 bytes, and is accordingly handled by the **_inmess** and **_outmess** routines. It is not necessary to send the size of the array with the transmission, because the C destination must know this in order to declare a suitable amount of local environment storage (achieved using the **#define** usage described earlier).

The C sub-ordinate destination may contain the following :

```
main( ... )
{
    double dbl_array[SIZE_VEC]; /** local storage **/

    for (i=0; i<SIZE_VEC; i++)
        _inmess(out[2], dbl_array[i], 8);

    F1 (dbl_array);

    for (i=0; i<SIZE_VEC; i++)
        _outmess(out[2], dbl_array[i], 8);
}
```

If the size of the array were *not* specified in a **#define** statement, then it is necessary for the stub message protocol to include the number of elements being transported at each useage – as an additional parameter. Ensure that a sufficient maximum local memory is declared to accommodate the biggest ever array transfer.

An observation on efficiency is appropriate. All array elements are stored contiguously. If the sub-ordinate and super-ordinate modules allocate array storage “compatibly”, then it would be advisable to block-send the appropriate number of bytes occupied by the array, in one operation. Again, this avoids computation and looping overheads. This is straight-forward if the two modules are written in the same language. If this is not the case, differences in the number of bytes per element, array index subscripting, and multi-dimensional storage must be accommodated.

If a pointer happens to point to an actual function rather than data (as permitted in C for example), then this would be a good instance of including that function in the *same* module as the one that references it by a pointer, rather than in different modules.

To avoid the overheads of sending strings and arrays as messages, a short-cut is possible *if it is guaranteed that the participating modules are resident on the same transputer*. In this case, the start address of the data area may be passed a parameter into another module. This approach can be used in any language, because it is possible to call a C function to determine the address of any data item. Remember — if this approach is used and the modules execute on *different* transputers, the results will be interesting to say the least.

10.5.5 Software methods of increasing performance

Once one has a system operating, there are several areas one can explore to increase the system performance, for example, by a greater overlapping execution of interacting modules. Some ideas are explored in this section.

Good ideas

This section lists a few useful techniques for increasing performance in a module-based system.

- **Using two stubs per module**

The use of a single stub to call a module means that explicit execution overlapping between the super-ordinate module and the sub-ordinate module is not possible — the super-ordinate sends messages to the sub-ordinate module then deschedules until the sub-ordinate completes. At the expense of including some additional synchronization code to the application, two stubs per function entry in a module can be used.

The existing function stub in the super-ordinate module is split into two parts. The outgoing message protocol goes in a “start stub”, having the name originally used to access the module. The incoming messages go in an “end stub”, with a (uniformly) slightly different but meaningful name. The programmer inserts a “call” to the end stub at the latest possible moment in the super-ordinate, following the start call, which serves to prevent the super-ordinate doing anything with results or changed parameters from the sub-ordinate. This allows the super-ordinate to continue processing

while the sub-ordinate executes. The performance increase is most apparent when the modules are on separate transputers. The re-synchronization is provided by the OCCAM channel mechanisms. One then still retains all the stub advantages and also has the capability to overlap executions. This is shown in Figure 10.13. Note that evaluating 6 and 7 does not depend on results of 3, 4, or 5.

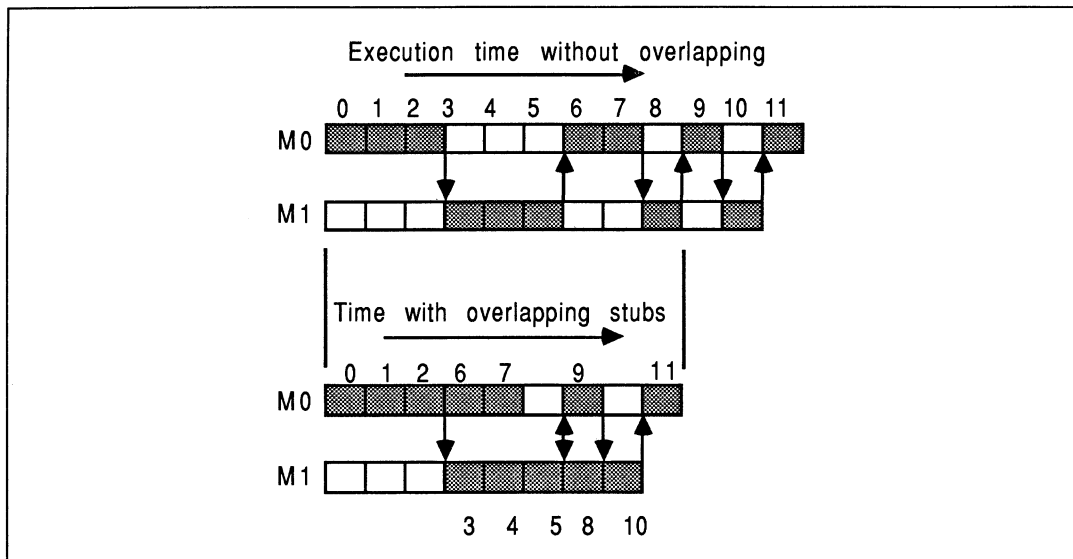


Figure 10.13 Overlapping execution using two stubs per module

Not every function entry point in a module need be converted to two-stubs, and of course neither must every module. The stub technique is a convenient way to achieve this.

- **Send results before house-keeping**

The implementation of modules often allows a previously unexploited execution overlap between communicating modules.

For example, a sub-ordinate module could return all results to the super-ordinate module, and *then* do any house-keeping and re-initialization. In cases where a call to a particular function in a module does not return anything, but just cause some action to be done on a data structure, there is no need to re-synchronize with the super-ordinate module.

- **Byte vector communications**

If the module access protocol involves many, many wonderful messages, it may be more efficient to package them all up into a byte vector and communicate that in one go. This approach requires that both partners undertake some encoding / decoding of data items. However, since each communication has an associated small overhead in setting it up, regardless of the size of the communication, these overheads are significantly reduced. A further advantage is that this type of protocol is easy to route in farming situations. If the vectors are too large (more than a few Kbytes) then latency penalties may be unacceptable.

Another approach would be to look at the division of the application into modules, since module should be selected so as to be compute-intensive with relatively low inter-module traffic.

- **Mingling communications and computation in the sub-ordinate**

In the sub-ordinate module (containing the actual bodies of the function grouping being stubbed), it would be possible to have the message-passing inputs interleaved with calculation (once the identity of the requested service had been established). Similarly, some results may be returnable before the end of the module's work. If the message protocol is carefully chosen, this can be intermingled with output calculations.

The extent of mingling in either of the above cases requires the programmer to make decisions about the earliest moments that communications can be performed. The modifications to the structure of the functions in the sub-ordinate require to be done only once.

For example, consider a function FX with six parameters and a return value. The first three parameters are part of the module access protocol, the last three and a result are output only. The sub-ordinate scenario might look like this :

```

while (running)      /** main loop **/
{
    ... receive command tag from super-o
    if (tag == DO_FX)
    {
        ... read a, b, c from super-o
        result = FX(a, b, c, &d, &e, &f)
        ... write d, e, f, result to super-o
    }
    ... else other things
}
... source for FX(a, b, c, d, e, f)

```

By having the protocol changed between the super-ordinate and the sub-ordinate, to facilitate maximum overlap in execution between the two participating modules, the message reception can be interleaved with the body of FX. The body of FX becomes :

```

int FX()
{
    ... get a from super-o
    ... do calculations
    ... get c,b from super-o
    ... do lots of calculations
    ... send f to super-o
    ... do even more calculations
    ... send d, result, and e to super-o
}

```

This technique will usually result in parameters being transferred in a different order to that in which the function is instanced with.

Bad ideas

These ideas are fast, but lead to loss of structure and lose some (or all) of the portability aspects offered by stubs. Basically, these approaches should only be used when one is *certain* that the communications protocols won't change between neighbours. The bullets are in order of increasing nastiness!

- **Unbunched in-line code substitution in the super-ordinate**

Using a knowledge of when certain items of data are available, it is possible to replace sub-ordinate module access stubs by in-line messaging primitives in the super-ordinate module. This involves sending parameters as soon as they are available, and may even involve interleaving accesses to several sub-ordinate modules at the same time. For maximum effectiveness in module overlap, an in-line code substitution for the message-handling body of each stub must be done, *where ever the stub is referenced* (ie frequently). This can be macro-automated using an interesting text editor, to minimize risk of error. Then, each messaging primitive is pushed as far down the source code as

possible, thereby increasing potential module overlap.

While this approach (coupled with the others) introduces the maximum possible opportunity for parallelism in a system (without introducing new modules), it is desparately difficult to modify the communications protocol between neighbouring modules. Also, the likelihood of errors and deadlock introduced by unsuspecting subtle modifications to the logic of the application code is increased.

For example, referring to the use of FX in the super-ordinate module, a stub would normally be used to ensure all references to FX conformed to the correct protocol. The FX stub would be frequently used within the module :

```

while (running)      /** super-o **/
{
    ... chuff
    result = FX (a, b, c, &d, &e, &f)
    ... chuff chuff
    result = FX (a, b, c, &d, &e, &f)
    ... chuff chuff chuff
    result = FX (a, b, c, &d, &e, &f)
}

```

Using the technique just described would result in this :

```

while (running)      /** super-o **/
{
    ... chuff
    ... send a to sub-o
    ... mingled chuff on outgoing
    ... send c, b to sub-o
    ... mingled chuff not involving FX
    ... receive f from sub-o
    ... mingled chuff on incoming
    ... mingled chuff not involving FX
    ... receive d, result from sub-o
    ... mingled chuff on incoming
    ... receive e from sub-o
    /* we've just done ONE access to FX */

    ... do other two accesses in the same way
}

```

The performance increase at this stage is overshadowed in comparison to the problems that could be introduced. All appeal from the standpoint of maintaining a clear structure and the ease of changing protocols is lost. It's a lot of work to implement and the code size penalty can be large.

Don't do it!!

• Making assumptions about data storage

By making assumptions about the way the compilers store data, it is possible to communicate many discrete values as byte vectors and save on the encoding / decoding of parameters. This is done by taking the address of the first of a list of variables, and sending a certain number of bytes of data from that address. The assumptions are that the variables to be transmitted occupy consecutive byte locations (yet could be of mixed type). Remember that currently, successive array elements occupy ascending memory addresses and are allocated from the module's heap storage [3]. Further, by declaring the data items identically in the destination module, it is possible to stream the byte vector into memory used by the required variables.

This is a very dangerous technique, which is very dependent on the current revision of the compiler being used — INMOS make no guarantees about the persistence of storage allocation strategies between tool releases. This is safest if participating modules are implemented in the same language. Yet, in some situations this is a useful technique.

10.5.6 Further work

Only after a single-transputer working system is demonstrable, one may elect to implement the following :

- Optimize the performance by ensuring that the development tools are being used to the best advantage, in terms of using on-chip RAM for the stacks of the most compute-intensive modules, and also in terms of code-utilization. Refer to [3] for guidelines.
- Incorporate additional processors into the system. Considerations of link interconnectivity, processor loading, and memory availability [8] can be used to guide the distribution of modules over various hardware topologies. This distribution has a dramatic effect on the overall system performance.
- One can further optimize system performance by injecting hardware compute power just where it's needed in a transputer network. The pin-compatibility between the IMS T800 and IMS T414 makes it straight-forward to have a mixed-processor network on non-TRAM boards like the INMOS B003. The range of INMOS TRAM modules [4] offers a selection of processors, speeds, and memory configurations which can be tailored to the requirements of the application.
- Retro-parallelize existing modules into finer-granularity systems, following careful study of the computation and communication requirements of the module concerned.
- Introduce multiple-stub techniques at strategic places to obtain more processing overlap.

10.6 Using transputers with other processors

In some applications, it is advisable to consider retaining an original processor in addition to using a transputer. This is known as a part port. In some cases, the other processor will be the host development platform, and in other cases the target environment will be a custom processor card. For the remainder of the discussions on using transputers with other processors, the word *processor* will be used to refer to a non-transputer processor. The words *host* and *target* will be used interchangeably and without loss of generality for the remainder of this document, to indicate processors operating symbiotically with a transputer (network).

Figure 10.14 shows an arbitrary processor which interacts with a range of devices and peripherals. A layered software structure is shown.

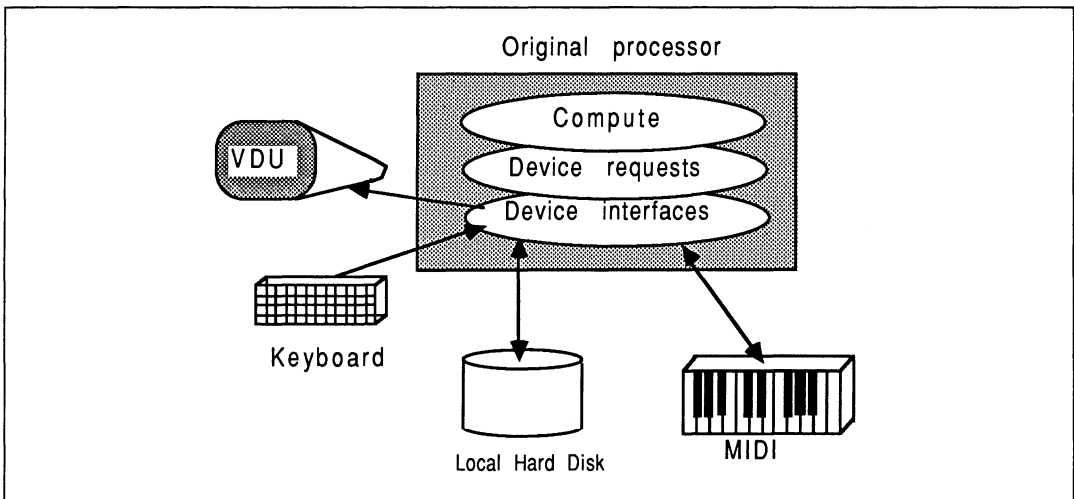


Figure 10.14 Before a mixed-processor porting

The lowest-level device driver interfaces operate intimately with the hardware of the original processor. The

computation part of the application is quite separate from the low-level interfacing code, and uses clearly identifiable requests to the device handlers to obtain the required device services. The application is independent of the implementation details of these devices.

The software situation following a part port is shown in Figure 10.15. The non-transputer part still handles the machine-dependent interfacing, and need hardly be changed. This saves time in putting together a mixed processor implementation. The lowest-level device handlers are left alone, and the computation parts are replaced by stubs. Before, device requests from the computation part of the application were directly handled by the low-level device drivers. Now, device requests from the computation elements on the transputer pass through a communications medium (a transputer link) and are reproduced exactly on the other processor using the stubs.

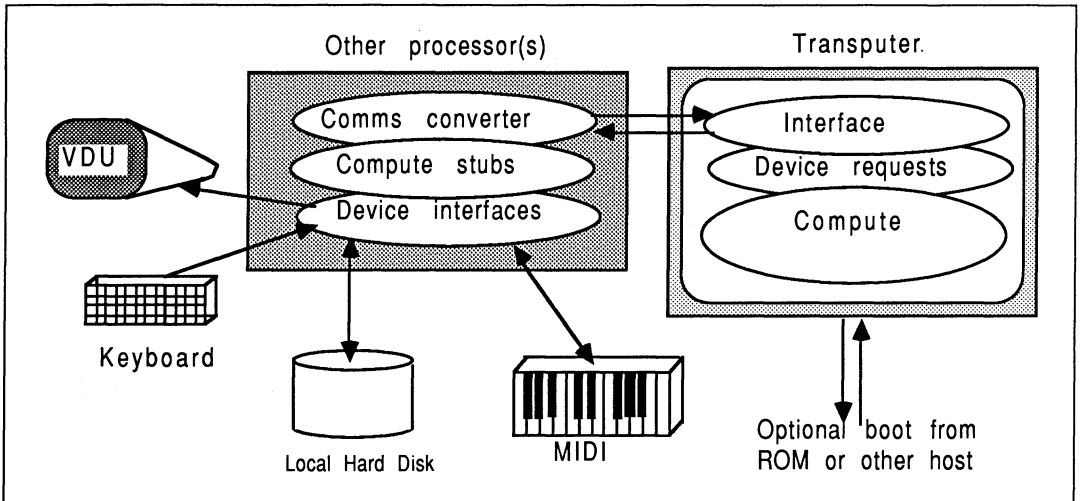


Figure 10.15 After the mixed-processor porting

The part of the application that is implemented on a transputer is subject to all the previous considerations and techniques in the previous section, on the subject of parallelizing an application.

10.6.1 Suitable applications

Retaining another processor in a system is sensible in applications that make intricate use of some target-dependent facilities, yet are heavily computationally intensive in some clearly identifiable areas. This is often the case in embedded systems, where the transputer can offer invaluable performance rewards.

For example, the following cases could be considered as suitable for partly leaving on host/target hardware in a mixed processor environment :

- If the application has normal but heavy host i/o references scattered unclearly throughout it — putting *all* this on a transputer and parallelizing it would involve through-routing overheads in multiplexing to the host server, and bottle-necking at the transputer's link adapter. In this case, leave all the heavy i/o on the original host — don't give data further to travel by placing code on a (distant) transputer.
- If part of application is written in a language other than C, Pascal, FORTRAN, or occam eg, 80286 assembler (or any other non-supported language), then clearly it has to stay where it is. This is also true if some source is not available at all, or is proprietary to another company, or would compromise maintenance and update arrangements, or is under the jurisdiction of another vendor. In these cases, don't move the affected parts from their original host.
- If the application makes intimate assumptions about target environment interfaces, peripherals, or specialized hardware, (such as DMA⁵ controllers, blitters, SCSI interfaces, memory-mapped graphics, VME devices...) then *leave it!* Software that talks directly to hardware is notoriously difficult to write, debug, and is usually timing-critical. It would be inadvisable to disturb such software, so only move the hardware-independent parts that need some compute-power.

Envisage a system solution which retains the specialist hardware products currently in use, yet employs a transputer network to increase performance where its needed. This is certainly to be advised as a first approach, as the existing hardware investment and interfacing which already operates, is guaranteed to be working.

Some examples of applications which may contain segments to be left undisturbed are those including :

- Acoustics : from simple host "bleeps" to advanced multi-instrument control using MIDI⁶.
- Laboratory instrument control and monitoring using IEEE 488⁷.
- A memory-mapped menu-based graphics user-interface system, with a pointer device for inputting commands (in addition to the keyboard). For speed, this would directly write bytes into host's screen RAM. Such WIMP⁸ applications are very attractive to use, and are normally a friendly interface to something computationally heavy. For example, the type of WIMP interface presented by Turbo C and Turbo Pascal would be ideal for leaving on the host.

Many non-WIMP user interfaces are assembler-based for responsiveness. For example, the Lotus 1-2-3 spreadsheet user interface. If an application like Lotus were to be ported to transputers, the fast user interface would stay on the host, and the computationally intensive parts would be run on transputers. The result would run faster than implementing the whole application on a transputer, because of the communication overheads between the host and the transputer network. For example, this technique was adopted in Risk Decisions' "Predict" Monte-Carlo simulation package.

Working with other processors requires awareness of a few architectural differences that can exist between these processors and the transputer network.

⁵DMA — Direct Memory Access.

⁶MIDI — Musical Instrument Digital Interface — an industry standard for interconnecting musical equipment.

⁷The General Purpose Interface Bus (GPIB), sometimes known as the Hewlett-Packard Bus — an industry standard for laboratory equipment interfacing and interconnection.

⁸WIMP— Windows, Icons, Mice, and Pull-down menus.

10.6.2 Software support for mixed processor systems

In dealing with mixed processor systems, some software support is required to handle problems presented by the interactions of several heterogeneous processors.

Accommodating architectural differences

When dealing with other processors, certain inevitable differences between the machine architectures have to be allowed for. These differences are most clearly handled by the code at each end of the communication link between the target/host and the root transputer, directly involved with data interchanges. This has the effect of minimizing the amount of code that has to accommodate these differences. The differences in question include the following :

- Differences in word lengths. Generally, the other processor will not be a 32-bit word length machine.
- The compilers on both machines will probably allocate different numbers of bytes for certain fundamental data types. As an example of this, an IBM PC with MicroSoft C will allocate 16 bits for an integer, whereas the transputer C compiler will allocate 32 bits. Therefore, in transferring integers between the host, this difference must be accommodated.
- Differences in byte ordering within words.
- Differences in bit ordering within bytes.
- Differences in floating-point representation. Any two *software* implementations of floating-point support are almost guaranteed to be different. Most software implementations do not conform to the ANSI/IEEE 754-1985 floating-point standard, with which all INMOS software complies. For example, real numbers in Turbo Pascal on a PC occupy 6 bytes, (unless a floating-point co-processor chip is available for the PC). Therefore, to transfer real numbers between the PC and the transputer, 4 or 8 bytes (for single or double precision) must be derived in IEEE 754-1985 format for the transputer — a similar conversion must be performed to exchange data the other way round.

Caplin Cybernetics offer a VAX/CSP library package to automatically handle data exchange and format conversion between MicroVAX and a transputer network.

Using services provided by another processor

There are some implications concerning the software support required on both processor types if code on one processor references something provided by the other processor. For example, consider the transputer-host relationship :

If the transputer requires any file, screen, or keyboard operations, obtained by calling standard run-time library functions, then host software must be capable of supporting the standard server protocol in addition to any other programmer-defined protocols required by the application⁹. This implies that, during this time at least, the host software assumes a slave role to the transputer. If this is always the case, then one way to implement this while taking advantage of existing INMOS software is to embody the application within the standard server structure, and simply add any new protocols to the repertoire. This has the further advantage of providing a capability to boot the transputer code *and* "serve" it in one neat manoeuvre. However, if the host code is normally the master of the pair, then it may be easier to embody all relevant parts of the server (to provide protocol support) into the host part of application (rather than the other way round which has just been described). In this scheme, the host part will temporarily delegate master status to the transputer while the server communications are under way.

In the relationship between the transputer and other non-host processors, a custom protocol must be devised to allow the partners to request services provided by the other. As long as all partners are kept synchronized to the extent of one master and one slave at any one time for any one communication, there should be no problems. Once this synchronization is achieved, a simple stub technique like that used on a wholly-transputer system offers little disruption to the actual operation of the functions being performed remotely.

⁹The run-time library translates anything requiring host services into a protocol sequence, and sends this protocol to the host server.

10.6.3 Hardware support for mixed processor systems

The interface between a transputer network and any other processor is normally achieved by using an INMOS link adapter¹⁰. This is a byte-wide peripheral, that is memory-mapped into the i/o address space of the other processor. It can be read from or written to by the other processor, in the same way as for other peripherals.

The first step to implementing a transputer to any foreign host would be to establish operation of a link adapter. This then permits both processor types to communicate. The rest involves writing software to exchange data meaningfully, bearing in mind the architectural differences that may exist.

Depending on the total system scenario, it may be necessary to permit the other processor to reset or analyse the transputer at will, or to boot code into it, or monitor the error flag¹¹. These supervisor functions are most easily accommodated by mapping in an 8-bit register (in addition to the link adapter), at a different address of course. Reference to any INMOS board-product documentation will demonstrate the relevant hardware techniques. Conformity to the INMOS techniques for implementing link adapter / system supervisory services is advised.

At application run-time, the transputer code must be booted onto the network. If the development host is part of the run-time configuration, then the host can be used to boot the network. The standard toolset server can be used to boot the transputer network, without the need for the programmer to build code into the host part of the application to fulfill this requirement. The host part of the application can then be started up. For example, to boot the transputer with `transbit.bt1` associated with `hostbit.exe`, the following simple DOS batch file could be used :

```
iserver /sc transbit.bt1 /sr /s1 #300
hostbit
```

This has the effect of resetting the root transputer and copying the boot file to it, using the link adapter at address hex 300. In addition to booting the transputer, the target part may also be used to monitor the transputer error flag.

Alternatively, the transputer code can be booted from ROM; this is probably the preferred option in embedded systems. Previously, the technique here was to have the root transputer explicitly boot from ROM by tying the transputer's **BootFromRom** pin high. This places certain requirements on where the ROM has to be in the transputer's memory map, and adds design complexity to the external memory system. However, it is now possible to have a transputer network boot from ROM by streaming the network boot data in any link on the root transputer. Phil Atkin's published design of a suitable TRAM, called the *TransBooter*, is given in [9]. As well as simplifying the system design by allowing off-the-shelf only-RAM TRAM boards to be used, the *transBooter* TRAM also allows fewer ROMs to be employed than would be required in a direct boot from ROM situation (because it doesn't require word-wide ROMs; only byte-wide), it also overcomes the 64 Kbyte limitation of a 16-bit transputer based **BootFromRom** system by allowing up to 512 Kbytes of EPROM.

¹⁰In some cases it is legitimate to memory-map other devices into the address space of the transputer, rather than the other way round as is discussed here. This is not normally done for other processors, but usually only for peripheral devices having a small number of registers and high requirements for data exchange rate.

¹¹When dealing with multiple transputer solutions, there are opportunities for allowing some transputers to have reset control over others. This hierarchy is implemented using the INMOS triplet of subsystem control ports *Up*, *Down* and *Subsystem*. Refer to any INMOS board product documentation.

Depending on the physical arrangement of the mixed processor system, some special considerations may be appropriate. The transputer links should be buffered if used between equipment racks, or where the electrical environment is "noisy". RS-422 differential transceivers can be used reliably at up to 20 Mbits/second over respectable distances (tens of metres). Fibre-optic communications products should be considered where electrical isolation or greater distances are involved.

As far as the transputer software is concerned, a link is a DMA engine. The other processor(s) can use this to advantage if they have the necessary hardware to support this (ie, a DMA controller), or they can simply use "busy" polling techniques to exchange data. There is clearly a performance implication here, which needs some discussion. The subject of heterogeneous processor communication is explored further in the next section.

10.6.4 Communication mechanisms

The transputer communicates with other processors by reading and writing messages down the links. This is how the transputer communicates with the development host, for example. There are several different ways to handle this communication. In each case, any architectural differences between the communicating processors have to be accommodated. Some of these communication methods are described now.

Communication by explicit polling

This is the easiest method of communicating between the transputer and another processor. Transputer development boards such as the IMS B004 are communicated with by a host server program using "polling". While this is the simplest method for the host to communicate, it is also the slowest. This is because every byte is explicitly transferred and a polled handshaking is in operation (a particular bit a status register port). The ease of accessing port addresses depends on the compiler's implementation of low-level facilities for the language in question. Some source examples in C and Turbo Pascal of poll-based communication are now given, for use on any processor other than the transputers.

- C communications

The following examples of C host-source for communicating with the link adaptor are taken from INMOS server source :

```

#define BIT_0 1

int byte_from_link ()
/* Read a byte from the link adaptor. */
{
    while (!(inp(link_in_status) & BIT_0))
        ;
    return (inp(link_read));
}

void byte_to_link (ch)
int ch;
/* Write a byte to the link adaptor. */
{
    while (!(inp(link_out_status) & BIT_0))
        ;
    outp (link_write, ch);
}

int word_from_link ()
/* Read a word from the link adaptor. */
/* A host INT is 2 bytes, and a transputer INT is 4 bytes.*/
{
    register int t, ch;
    t = byte_from_link(); /* 1.s. byte */
    ch = byte_from_link(); /* m.s. byte */
    t = t | (ch<<8);
    ch = byte_from_link(); /* Ignore upper 16-bits */
    ch = byte_from_link(); /* sent by transputer */
    return (t);
}

void word_to_link (w)
/* Write a word to the link, least significant byte first. */
/* A host INT is 2 bytes, and a transputer INT is 4 bytes. */
int w;
{
    byte_to_link (w & 0xff); /* 1.s. byte */
    byte_to_link ((w >> 8) & 0xff); /* m.s. byte */
    if (w < 0)
    {
        byte_to_link (0xff);
        byte_to_link (0xff);
    } else
    {
        byte_to_link (0);
        byte_to_link (0);
    }
}

```

Notice that the last two functions, `word_from_link` and `word_to_link`, accommodate the architectural differences between the transputer and the host, in terms of word length and endian considerations.

- Pascal communications

The following examples of Pascal host-source for communicating with the link adapter are taken from an example shipped with the IMS B008 PC-format TRAM motherboard :

```

const
  linkBaseAddress = $150;
  inputData       = 0;
  outputData      = 1;
  inputStatus     = 2;
  outputStatus    = 3;

procedure outByte (b : integer);
begin
  while not odd (port[linkBaseAddress + outputStatus]) do
  begin
    { do nothing }
  end;
  port[linkBaseAddress + outputData] := b;
end;

function inByte : integer;
begin
  while not odd(port[linkBaseAddress + inputStatus]) do
  begin
    { nothing }
  end;
  inByte := port[linkBaseAddress + inputData];
end;

procedure outWord (w : integer);
begin
  outByte (w AND $FF);
  outByte ((w SHR 8) AND $FF);
  if w < 0
  then
    begin
      outByte ($FF);
      outByte ($FF);
    end
  else
    begin
      outByte (0);
      outByte (0);
    end;
end;

function inWord : integer;
var
  b0,b1,junk : integer;
begin
  b0 := inByte;
  b1 := inByte;
  junk := inByte;
  junk := inByte;
  inWord := b0 + (b1 shl 8);
end;

```

Again, the architectural differences are accommodated in these low-level routines. Once written and tested, all communications use these routines. Architectural dependencies are localized.

In communications-limited situations, if several link adapters are available, then it is possible for the other processor to send bytes to each one cyclically (ie, round-robin), which can almost linearly increase the data transfer rate. This is because time which was previously dead-time is now used productively, and once polling begins there is much less time to wait before one can proceed with the next transfer.

For large amounts of traffic, the "status register polling" technique can be more time consuming than necessary, even with additional link adapters. An alternative technique is to use DMA.

Communication by explicit DMA

Some INMOS development boards (such as the IMS B008) support a more complex method of data transfer, well suited to moving blocks of memory at high speed, between the host and the transputer. This method is known as DMA (Direct Memory Access), and can be used to achieve higher performance in communication-limited situations between the host and transputer (network). It operates by freeing the other processor of the supervising of data transfer, allowing it to do other things in the meantime.

DMA is particularly well suited for large blocks of memory, such as image data perhaps, being transferred from a device on the PC bus to the transputer for processing. This is because the actual data transfer is performed by additional hardware circuitry which supervises the transfer independently of activity on the main host processor. DMA can also be used between the transputer and any other processor in the system, providing each processor concerned is equipped with a DMA controller.

The initialization of a DMA transfer is fully dependent on the DMA controller chip used by each participating processor. It typically involves setting up a few memory-mapped registers, used by the DMA controller, to specify the direction of DMA (ie, from the transputer to another processor, or the other way round), the number of bytes to transfer, and the address. The memory transfer is then handled invisibly by the DMA controllers. Again, architectural differences between the processor types have to be accommodated.

The performance improvement over the simple polling technique is typically a factor of two, rising for large block movements.

Communication by device drivers

Although this technique is not likely to be applicable to an embedded system, a processor which runs a full general-purpose operating system can offer the previous techniques of polling and DMA transfer *implicitly*, without the application having to be aware of the mechanisms employed. Assume that the application runs partly on such a processor, and partly on a transputer system.

Many operating systems allow the programmer to add device handling capabilities, known as *Device Drivers*. They allow any program running on the machine to access the device via the operating system. By using a device driver to handle communications, programs can gain access to the transputer card in an efficient way, without needing to know anything about the specific hardware interface. Caplin Cybernetics implement communication between a transputer network and a MicroVAX using device driver techniques.

In UNIX on a Sun-3, adding new device drivers requires the operating system kernel to be recompiled and linked with the new device driver. In MS-DOS, all device drivers are installed at boot-time, which makes adding new ones simple. There are two distinct kinds of device under MS-DOS :

- **Block Devices** : These are devices which usually support a disk file structure. A block device driver tells MS-DOS about the physical characteristics of its device (Block Size, Maximum Device Capacity *etc*) at start up time and thereafter DOS will only make calls to read and write blocks on the device. A block device cannot be opened as if it were a file because it is itself a *File System*. In this way only MS-DOS can access the device directly, the application program may only access the device indirectly through files held on it.
- **Character Devices** : These devices have support for transferring random amounts of data to and from the device. Unlike block devices, character devices can be opened as if they were files. This allows data to be transferred to and from the device using the normal MS-DOS read and write to file calls.

If an MS-DOS character device driver [10] is installed on the PC to service the link adapter, then the host part

of the application can communicate with the transputer simply by opening a "file" and reading / writing data in the normal manner. The transputer software would read and writes bytes to and from the link connected to the host, and form these bytes into the correct numeric representations.

For example, the following fragment of host Turbo Pascal attempts to open a device driver installed under the name of **IMS_B004** for reading and writing data to the transputer software :

```

program DeviceTalkerToTransputer (input, output);
var
  toLink, fromLink : text ;
  ... procedure definitions
begin
  writeln ('Device Driven communications');
  if fopen (toLink, 'IMS_B004', 'w') and
    fopen (fromLink, 'IMS_B004', 'r') then
    begin
    ... initialize link adapter and transputer
    ... perform comms using toLink and fromLink
    end
  else
    writeln ('Error opening link device driver')
  end.

```

In a PC system, the list of device drivers is specified in the **config.sys** file. For the link adapter device driver, the configuration file might have this line in it :

```
device = c:\bin\linkdriver.sys 150 160 IMS_B004
```

The **linkdriver.sys** file is the device driver image, **150** and **160** represent the link adapter reset and analyse addresses (on the host bus), and **IMS_B004** is the name of the device. The INMOS IMS B004 transputer evaluation board uses addresses #150 and #160 on the host PC's bus. This technique allows the same device driver to be installed, with a different name, at a different address. So, for example, if the same PC had an IMS B008 card at address #300, then the **config.sys** file could also have this line :

```
device = c:\bin\linkdriver.sys 300 310 IMS_B008
```

To use this environment, the following host Turbo Pascal fragments read and write integers to the transputer :

```

var
  start, finish, i, j : integer;
begin
  ... initializations
  for i := start to finish do
    begin
    write ('sending ', i);
    writeln (toLink, i);
    readln (fromLink, j);
    writeln (' .. and received ', j);
    end
  end;

```

The communication of these integers must be handled on the transputer by software expecting to send and

receive bytes. So, for example, the following OCCAM PROC could be matched to the Turbo Pascal above :

```

PROC the.transputer.test (CHAN OF BYTE from.link,
                          CHAN OF BYTE to.link )
  #USE "string.lib"

  PROC readint (CHAN OF BYTE in, INT n)
    -- convert byte stream into transputer integer
    [100]BYTE string :
    INT str.l :
    BOOL err :
    SEQ
      GETSTRING (in, err, str.l, string)
      STRINGTOINT (err, n, [string FROM 0 FOR str.l])
    :

  PROC writeint (CHAN OF BYTE out, INT n)
    -- convert transputer integer to a byte stream
    [100]BYTE string :
    INT str.l :
    BOOL err :
    SEQ
      INTTOSTRING (str.l, string, n)
      SEQ i = 0 FOR str.l
        out ! BYTE string[i]
    :

  WHILE TRUE          -- main body
    INT data :
    SEQ
      readint (from.link, data)
      ... operate on data
      writeint (to.link, data)
      to.link ! '*n'
      to.link ! '*c'      -- allows readln in Pascal host
    :

```

The device drivers can be used from any host language supporting file access. By using device drivers and transferring information as "human-readable" quantities, all the architectural differences between the host and the transputer are nullified, as far as the application programs are concerned. So, considerations for endian, wordlength, and data-type representation incompatibilities are no longer important, as long as the transputer packages up the textual representation of data into suitable transputer format. This has considerable benefit in exchanging floating-point information, but there is an unavoidable inefficiency because more bytes have to be exchanged than for data encoded in the normal host representation.

A further benefit exists when it comes to testing the part ported system. It is possible to test one part without the other, by using *real* files containing dummy information which represents the data being trans-communicated.

For example, to test the host input and output messaging under a specific execution path, an ASCII file would be prepared containing expected responses from the transputer system. The host would open that file for input in place of the incoming channel from the link adapter. The resulting output communications from the host part can be fed to file and analysed afterwards. Because all data is human-readable, it is easy to prepare and check test cases.

Increasing data exchange bandwidth by software means

If the hardware implementation of a communications interface between a transputer system and another processor is most efficient at exchanging "large" data blocks (100 to 10000 bytes say), then performing mostly single-byte transfers is far from ideal.

To take best advantage of this, it may be necessary to arrange for the application to perform mostly multiple-byte transfers. For example, a typical file loading would sequentially read bytes from a file until the end was reached. By either placing a filter on the channels between the application and the file store, or by adjusting the application slightly, large data blocks can be requested for transfer. The situation is depicted in Figure 10.16

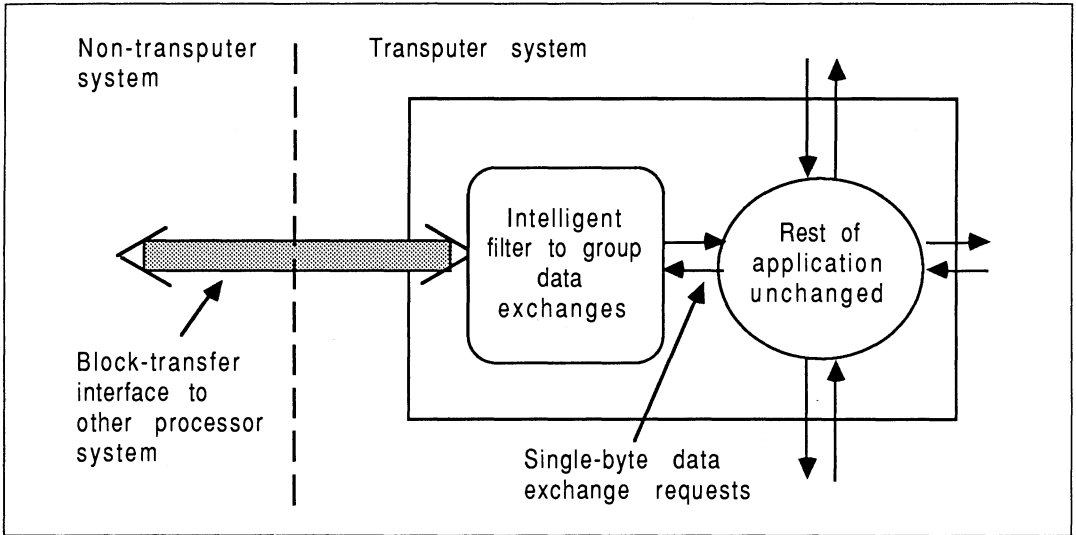


Figure 10.16 Using a filter to block-up data exchange requests

The filter technique is application independent. It consists of a software process on the transputer connected to the other processor. The filter detects a request from the transputer application for a byte held (in a file perhaps) on the other processor, and delays servicing this until the next request has been examined. If the request is exactly the same, the filter blocks these up until (say) 128 byte reads (or writes) have been processed in succession. The filter then arranges for an appropriate interaction with the other processor in an efficient block transfer. If the request is different, it will purge its current backlog. This technique can be used to good advantage on DMA-type interfaces. The overheads in running the filter are dwarfed by the increase in performance due to efficient operation of the inter-processor communications.

10.6.5 Implementation strategy

The best implementation strategy for a mixed processor system is very similar to the methods of modularizing an application described in the previous section. The concept of using message passing stubs in the target parts, and unmodified original parts on the transputers, is unchanged.

- **Transputer code** : All transputer code should be implemented in one module on one transputer. This requires minimal OCCAM support [3]. Depending on whether host services are required, either the full or standalone run-time libraries are used. A tagged protocol between the transputer and the other processor is used to select one of several function groupings in the module. The module structure on the transputer is exactly the same as described in the previous section, making allowances for possible changes in the master / slave relationship.
- **The other processors** : All code now implemented on the transputer should be implemented as stubs on the other processors which access communication primitives to exchange information with the transputer (using the selected communications method). This localizes and confines all parts which communicate across the link adapter, and help to make the interface between the target and the transputer "clean". It also clearly minimizes the amount of recoding that would have to be done to re-implement them (the communications primitives) in target assembler for performance reasons. Ensure all communications use the *same* set of primitives, otherwise something mutually

incompatible is bound to show up at an inconvenient moment.

10.6.6 Testing strategy

There are several areas which could cause problems in a mixed processor system, so a phased testing and implementation is to be recommended. The following ordering is suggested :

- Initially, check that code can be booted to the transputer, and the exchange and modification of the fundamental data types (characters, integers, and real numbers etc) is operable. Initially this is done between any two adjacent hardware partners. This will also serve to test the software frameworks implemented on the transputer. Furthermore, any difficulties concerning floating point representation, differences in machine word length / endian possibilities etc will also show up. This test should be performed using some dummy functions, in a dummy module written explicitly for test purposes, but conforming to the ultimate module structure and using the ultimate OCCAM harness techniques.
- Once this has been shown to operate, the next stage is to implement *one* real function grouping (in only one module, and on only one transputer). This will test the proposed control structures operating between the two systems.
- Once operational, proceed to incrementally implement all the function groupings required, building on the *working* steps of the previous stages. This involves making stubs of the functions left on the other processors. Phase the stub implementation and test systematically as you go. Confine everything to one module for simplicity.

One unique feature of a system incorporating transputers is the ease by which the system can be investigated by software means when behaviour is unexpected. For example, simply connect any unused transputer link to a host platform supporting the *iserver*, (and arrange for the transputer to boot from link), and wheel in the toolsets' symbolic debugger. This allows an embedded system to be examined, or to have diagnostic code loaded instead, even if the debugger platform is not part of the normal configuration. A network memory dump can be taken at a remote customer site by a field engineer, and taken back to the laboratory for analysis and reproduction by the debugger.

10.6.7 Further work

Once the mixed processor system is operational, and all code intended for transputer targetting is implemented on the transputer, there are some other areas that can be looked into.

- Use the development tools to make best use of the transputer's on-chip RAM.
- Apply modularizing techniques from Section 10.4 to the transputer code.
- Then apply multiple transputers to the problem.

10.6.8 Mixed processor example

The implementation techniques discussed above are now illustrated by considering a Turbo Pascal application on a PC. Part of the application is to be ported and executed on a transputer, using the Transputer Pascal compiler.

Suppose only function **HugeTask** is to be ported to a transputer. It has two *value* integer parameters, and returns an integer result. It uses no free variables. The actions of the function are not important. The stub for this, left on the host PC, may look like this.

```
function HugeTask (a, b:integer) : integer; { stub !!}
begin
  outWord(OpCalc1);
  outWord(a);
  outWord(b);

  HugeTask := inWord;
end;
```

The integer tag **OpCalc1** is simply used to identify that the **HugeTask** function is required, rather than some other function which may be subsequently implemented. The procedure **outWord** and function **inWord** accommodate architectural differences between the PC and the transputer.

The Pascal on the transputer represents a module. A standard "entry-point handler" called **runController** is used to read the tag identifying the action requested, and then passes control to a message-passing procedure for the action; **DoHugeTask** in this case. This structure allows simple accommodation of many

different functions on the transputer :

```

module remote;

#include '\tplv2\channels.inc'

const
  OutChannel = 2;
  InChannel  = 2;

  OpCalc1    = 0;

function HugeTask (a, b:integer) : integer;
  { Define the original function here}

procedure DoHugeTask; { Message handler for HugeTask }
var
  a, b, result:integer; { local storage}
begin
  inmess(InChannel, a, 4);
  inmess(InChannel, b, 4);
  result := HugeTask (a, b);
  outmess(OutChannel, result, 4);
end;

procedure runController; { Standard component }
var
  stopping : boolean;
  command  : integer;
begin {runController}
  stopping := FALSE;
  repeat
    inmess(InChannel, command, 4);

    if (command = OpCalc1) then
      DoHugeTask    { List all module entry points here}
    else stopping := TRUE;

  until stopping;
end; { runController}

begin { main }
  runController;
end.

```

A standard OCCAM harness connects the Pascal module channels to the link adapter connected to the host.

The same structure of software components is used for any application, in any language. On the other processor, message-passing stubs replace the functions to be ported. On the transputer are placed all the ported functions (like **HugeTask**), a message-passing handler for each one (**DoHugeTask** performs communications to reproduce the original environment, calls **HugeTask**, and sends the results back), and one standard **runController** framework.

10.7 Farming an application

Farming involves using additional processing power (in the form of transputers) to work concurrently on a problem and thereby achieve a performance or through-put increase.

A processor farm consists of a transputer network, frequently of regular topology, which routes work to worker processors and retrieves the results when they are ready. The general farm structure is shown in Figure 10.17. It is independent of the application, and requires only minimal implementation modifications to a suitable modular application.

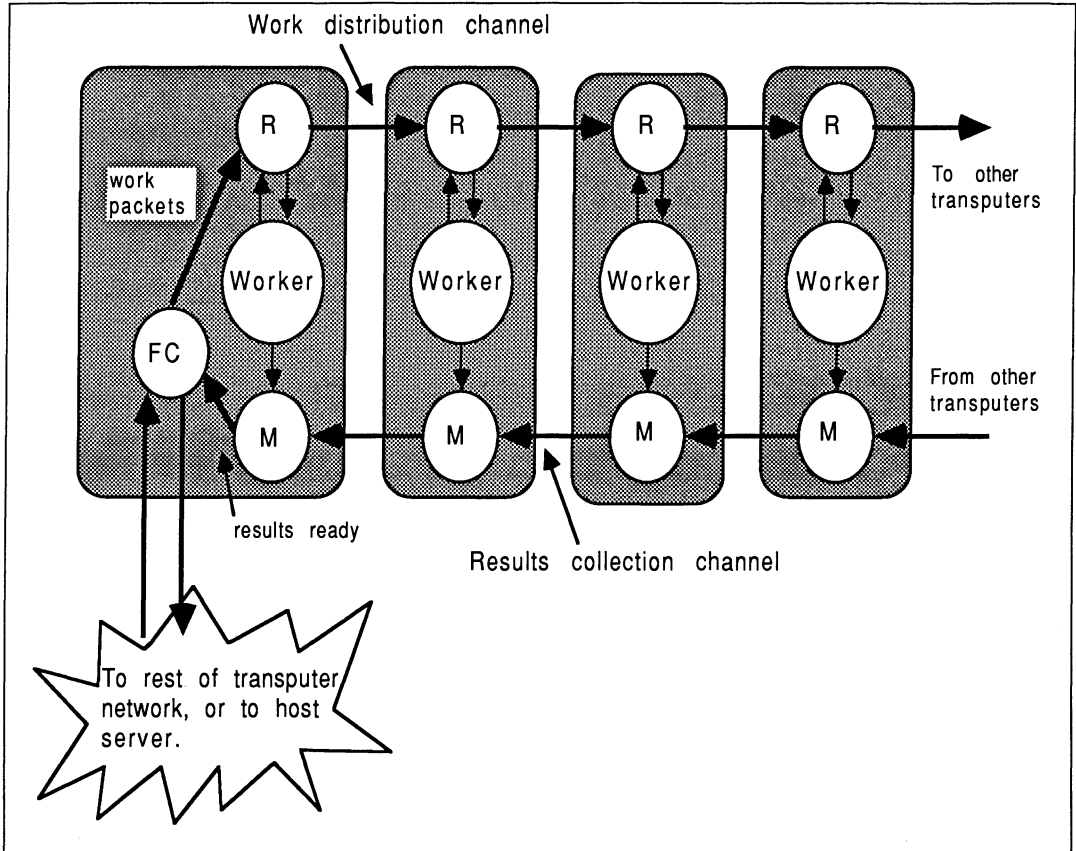


Figure 10.17 A general farm structure

In a processor farm, the work to be done for a single job is decomposed into a set of independent tasks which can be executed concurrently, with each processor performing a part of the total task. For example, the Mandelbrot Set [11,12] is a infamous example of a transputer farm. Each processor calculates a small part of a scene, which is gradually built up in patches. Only a small amount of input data is required to enable a worker to calculate the image for that patch of the total scene. Ray tracing [11,12] is another good opportunity for processor farming. A database of world objects is broadcast to all workers. Each processor then computes an image tile for a small segment of the image.

Alternatively, a farm of entire applications can be constructed. This is useful where the same application has to be run many times, each with different sets of input data, and where the outputs can be stored independently (perhaps on the host's filestore). Each transputer executes the entire application, which is completely unmodified.

Before looking in detail at some different categories of processor farm, consider what affects an application's suitability for farming.

10.7.1 Suitable applications

The requirements for introducing farming to an application include :

- Independent sets of input data are available (or can be determined automatically) at the start of execution.
- The application does not require user interaction to be associated with each data set while operating, ie, autonomous operation.
- The application is easily ported onto (initially) a single transputer. This involves satisfying the requirements given in Section 10.3.2.

10.7.2 General farm discussion

This section discusses farming in general, beginning with the various components found in a farm. Each transputer in the farm typically executes identical "worker" code.

The software components

There are three key software components in any farm, in addition to the application to be farmed. These are explained with reference to Figure 10.17.

- **Router process — R** : This process feeds work tasks into the farm. If the local worker application is busy, work is passed onto the neighbour processor.
- **Results merger process — M** : This process combines results from the local worker with those from the neighbour worker in the farm, and returns them towards the controller process.
- **Farm control process — FC** : This process controls the distribution of work and data around the system by sending jobs out whenever a processor is free, until no more are left. It never sends more work than there are workers. The control process closes down the farm once all the work has been done.

All these processes are written in OCCAM, but the application itself remains in C, Pascal, or FORTRAN. Each farm worker node requires a router and merger to be run in parallel with the application. The root transputer usually controls the farm, and therefore additionally executes the farm controller process, (although a farm can have it's root anywhere within a transputer network). Various data buffers employed for performance reasons are not shown in the diagram. Reference [11] gives good advice and many practical examples.

The farm protocol

All traffic within the farm should be made to conform to a rigid protocol. The design of the protocol is trivial in relation to the actual application, and can be made independent of the application. The usual procedure is for the protocol to allow control and data information to travel on the same channels. This protocol is typically variant, with less than five variants. In particular, a *Work Request* variant would be sent by the farm controller, and accepted by the first available processor; and a *Results statement* variant would be sent back to the farm controller from a worker. Other variants may allow debugging messages to be transported, and allow the farm to terminate cleanly.

In a farm, results often come back to the controller in a different order to which they were sent. This implies that with each results packet, information must be supplied to allow the non-farmed environment to receive the results and "put them in the correct place". A neat way to do this is to consider the specification of the work packet to include any necessary pointers / addresses useful to the receiving process. So, for example, any writable parameters, results, or free variables passed between the module to be farmed and it's calling environment, must *also* have storage addresses sent as part of the work request protocol. These addresses also form part of the results protocol, and allow the receiver outside the farm to store data in specific memory locations. An example of this is given later.

10.7.3 Interfacing to the farm

Usually only part of an application is farmed. This requires interfacing to the non-farmed bit. To achieve this, the farm control process has two channels to the outside world — one accepts work requests, and one delivers results. The OCCAM model of communication ensures synchronization between all the participating processes.

There are two cases worth looking at.

Interfacing to another transputer process

The **FarmInterface** process shown below is sent work requests on **CommandToFarm** by the rest of the application. Replies from the farm are collated and returned to the application on **FarmReply**. The farm protocol is defined as **farm.p**. Work is only injected to the farm if the controller has received a **t.ready** tag, indicating an available worker. The controller keeps a count of the number of available processors to which work has not yet been dispensed.

```
PROC FarmInterface (CHAN OF farm.p fromFarm, toFarm,
                  CHAN OF ANY CommandToFarm,
                  CHAN OF ANY FarmReply)

SEQ
  ... initialise
  WHILE farmActive
    PRI ALT
      -- Test for free workers needing tasks
      (FreeWorkers > 0) & CommandToFarm ? len::data
      SEQ
        ... send work to farm
        FreeWorkers := FreeWorkers - 1
      -- Handle data from farm
      fromFarm ? CASE
        t.ready; processor
        PRI ALT
          ... if work to send then send it
          ... remember available worker has no work
        t.results; processor; len::data
        FarmReply ! len::data    -- data to rest of appl.
    :
```

Interfacing to a process on a non-transputer processor

In the situation where work is being dispensed from a non-transputer processor, the synchronization offered by a purely transputer-based system is lost. The other processor, say the host, in order to synchronize with transputer activity, must be allowed to “free run” and send task after task to the farm before any results are received. This can be accommodated in a handshaking protocol between the host and the transputer, and encapsulated in a **HostInterface** process which connects directly to the host and the **FarmInterface** process.

The **HostInterface** implementation shown below operates as follows. If a valid job specification is received, it sends it onto the farm directly. If the farm is fully busy, it will be unable to receive any further correspondence from the host. If there are any results from the farm, they are returned as part of the handshaking; otherwise a null result is returned. At the end of the application, this mechanism will have resulted in fewer valid results received than job specifications issued. An enquiry tag **t.Enquiry** is used

to return results if there are any — this can be used by the host until all results have been received.

```
PROC HostInterface (CHAN OF HostToTP.p fromHost,
                  CHAN OF TPToHost.p toHost,
                  CHAN OF ANY CommandToFarm,
                  CHAN OF ANY FarmReply)

SEQ
  ... initialize
  WHILE going
    SEQ
      fromHost ? CASE
        -- Valid job for the farm
        t.ValidJob; len::data
        PAR
          CommandToFarm ! len::data
          PRI ALT
            FarmReply ? len2::data2
            toHost ! t.ValidResults; len2::data2
            TRUE & SKIP -- no results yet
            toHost ! t.NullResults -- null reply

        -- Farm enquiry
        t.Enquiry
        PRI ALT
          FarmReply ? len::data
          toHost ! t.ValidResults; len::data
          TRUE & SKIP -- no results
          toHost ! t.NullResults
    :

```

10.7.4 Performance issues

Linearity

Farming can offer a linear performance increase for an application. In some cases, this can be super-linear. For example, in the ray tracer farm, ten transputers can perform at eleven times that of one transputer. The reasons for this are two-fold. Firstly, the use of each transputer's on-chip RAM means that more of the "total" task can be accommodated in ultra-fast memory. Secondly, in any application there is generally an initialization portion which only has to be done once. In the ray-tracing case, the overheads of initialization are minimized because each processor does their own portion.

Priority

All routing processes are run at high priority to ensure that traffic is kept moving in the farm. This helps to ensure that workers at the extremities of the farm are kept serviced [13]. If the workspace of the routing processes is in internal on-chip RAM, the latency of response to link inputs is reduced¹².

The application code is executed at low priority.

Protocol

Counted byte vectors are frequently used within farms because this serves to make routing and merging simple, and is more efficient per byte sent — the amount of processing resource used by a communication depends more on the number of communications than the amount of data in each communication. However, for *large* byte vectors, there is a transfer latency penalty to be paid. By breaking a long message into shorter ones, several processors can transfer the data concurrently which reduces inter-processor latency (this is useful when broadcasting data throughout an array) [12].

¹²When a process is scheduled, several words are written into the workspace of the descheduled process. If this workspace is on-chip, the process swap-time is reduced.

Overheads

There is virtually no overhead in running the additional farm routing and control processes [11]. This is because the transputer can perform communication and computation concurrently and independently of each other. Processes waiting to communicate are descheduled by the transputer hardware, thereby freeing some processor resource for other processes. There is a penalty paid for setting up each communication. This is independent of the amount of data in the communication, favouring the use of efficient counted byte-vector communications protocols.

Buffering

Software buffers should be used to decouple communication and computation, allowing a greater overlap between them. In particular, buffers should be used on all routing and merging channels to decouple the communications from computation, allowing a very high compute-utilization [11, 12]. Copying data in buffers is inefficient, so a swing-buffer approach would be used — data fills one buffer while work is drawn from another, then the other buffer fills etc. This can have a dramatic effect on the overall system performance [11]. These buffers are easy to implement in OCCAM. For example, work packets can be pre-fetched and buffered locally by each worker transputer. This allows a new work task to begin immediately the old one is completed. Buffering on each input to the merger process allows a greater throughput, again by allowing a new task to begin immediately and achieve computation overlap with communication.

Load balancing

The farm achieves a run-time dynamic load balancing throughout itself, with the whole system keeping itself as busy as possible [13]. This is because workers accept work automatically as and when they finish a work packet. If the time taken to process a work packet is longer than the time taken to receive a new work packet from the controller, then each worker is automatically kept busy all the time. The end-latency in a dynamically load-balanced processor farm is much lower than in a statically load-balanced system.

General farming principles

The following list contains some additional points concerning processor farming :

- Farming is generally independent of the application, and if the entire application is being farmed it requires *absolutely no* modifications.
- Each worker activity operates on orthogonal (ie, independent) data sets with respect to other workers. A data set may correspond to an entire application's work information, or more probably it will represent a small proportion of a total task.
- The worker module should have preferably only one channel of "input" and one channel of "output" messages. If this is not the case, either modify the worker to satisfy this criterion, or make the Router and Merger processes reproduce the multi-channel environment locally on each worker transputer (otherwise the farm is not independent of the application).
- Farming is a particularly useful technique when the amount of computation required in any given sub-task is not constant [13].
- The work packet is devised to cause a suitable amount of work to be done by each worker, in relation to the overheads of routing the work request packets through the farm. This parameter is termed the ratio of computation to communication, and should be as large as possible.
- The ratio of computation to communication can vary with the hardware used. For example, [12] shows that introducing a floating-point transputer to a farm will drastically alter the computation load of a Mandelbrot Set worker. As a consequence, the size of work packet should be increased to retain a favourable ratio.
- Generally, farming implies that the same execution code is used for each worker transputer. The INMOS development tools create the smallest boot files if the code on each processor is exactly the same.

- In systems where some farm-wide read-only data has to be shared by all participating workers, (such as a physical model for a ray-tracer), this database can be broadcast once-only at start-up to minimize traffic per work packet during rendering requests.
- If smaller tasks can be sent to a farm towards the end of a job, this helps to keep *all* processors busy for the longest time. This is because all tasks take a different time to execute, when the jobs run out there will be successively more processors inactive before the farm shuts down. Saving short tasks until the end helps to minimize the amount of non-utilization.
- Farms arranged as a linear pipes are easiest to handle routing and distribution requirements. However, a structure having the minimum depth will result in the greatest performance because non-local communications slightly degrade performance.

Armed with this general farm information, it's time to look at three specific types of farm, and consider how they differ from the general farm discussion above.

10.7.5 Farming part of an application

Scenario

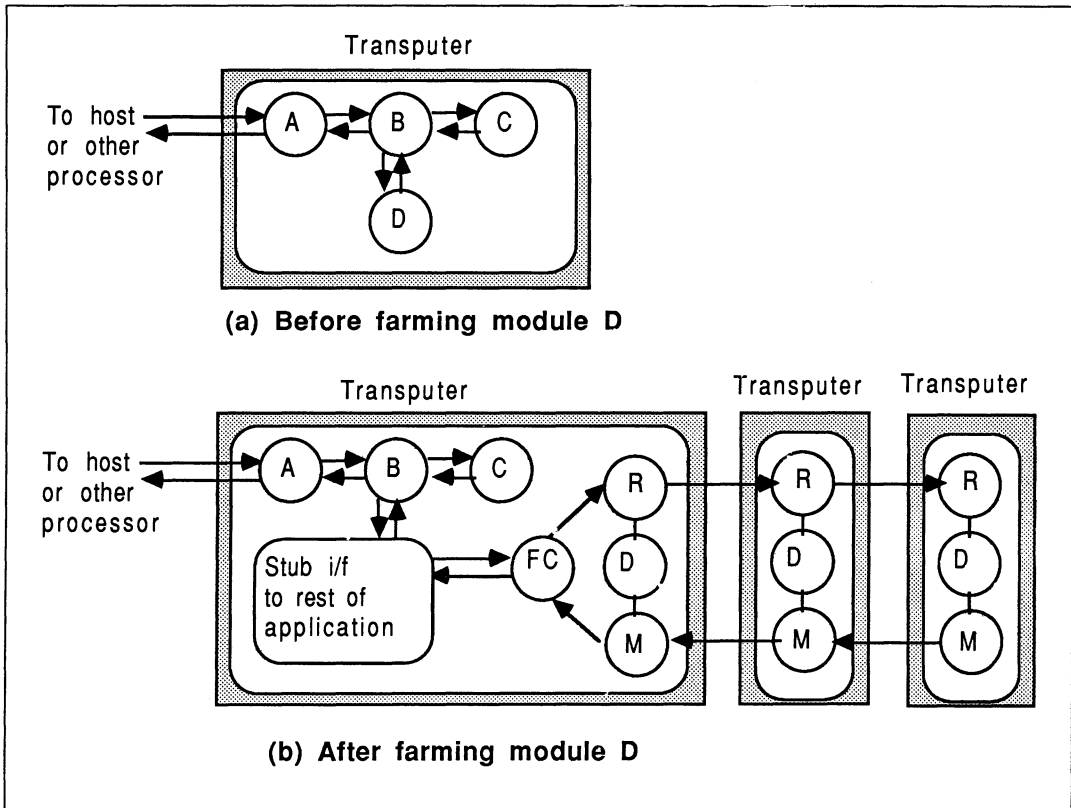


Figure 10.18 Farming part of an application

Start with an application that has already been ported to a transputer, and has been split into modules. Assume that a module can be assembled that represents the part of the application to be farmed (which is of course very compute intensive in relation to the amount of data that has to be provided with each work

task). This might be the case where a non-farmed user interface package dispenses work to a farmable computation module. It could also be a part ported application, providing that not *all* of the original transputer code is farmed. Figure 10.18(a) illustrates this situation, with Module **D** to be farmed. No assumptions are made about other processors the farmed module may require to interact with, but in this case the best results will come from farming a module that directly (or otherwise) makes no accesses to a host processor (because Module **D** is distant from the host and would require protocol routing on intermediate modules).

Implementation

The implementation starts with a ported application that has been decomposed into modules. One or more modules will be identified as the part of the application that is to be farmed — the worker; Module **D** in Figure 10.18(a).

With reference to Figure 10.18(b) : Between the non-farmed part of the application and the worker module, a farm control module, **FC**, is written in OCCAM. This communicates with the rest of the application via a stub, in such a way that the channels and protocols to the rest of the application are minimally modified. The farm controller divides the total task into a number of sub-tasks, and injects these into the farm using the Routers **R**. Module **D** is replicated in the farm, *completely* without change. Additionally, the farm controller **FC** ensures that results, which will be returned in a non-deterministic order via Mergers **M**, are suitably returned to the rest of the application (the results protocol will include enough information to allow them to be correctly slotted in).

10.7.6 Farming an entire application

Scenario

Consider farming an entire application, which is already successfully ported to a single transputer (this precludes a part ported application). The application uses host services, so it is necessary for these host accesses to be correctly interleaved to prevent cross-interference. The application is not modified — it doesn't even know it's in a farming environment. With reference to Figure 10.18(a), this would correspond to farming the whole of Modules **A**, **B**, **C**, and **D**, with Module **A** connected to the host. In this case, the controller **FC** of (b), instead of being a stub to the rest of the application, would enquire of the host the work to be done. The host will provide a list of (independent) tasks to be performed. Each transputer will perform an entire "application's worth" of work.

Implementation

This farm is implemented without modification to the application. Some issues are :

- **Task specification** : A file of tasks to be executed is provided by the user, and the farm controller reads this and distributes information to available processors. The lines of this task file are sent (usually) to appear as the command line parameters to the application programs running in the farm. These will typically represent the names of files which must be opened and used by each application — all the files must be different per application ! The rest of the command line parameters to the farm program could be appended and also sent to the farmed application.
- **Sharing server access** : Although the farm will support multiple applications accessing the server concurrently, the server facilities being used determine the suitability of the application for this type of farm. For instance, if the application participates in screen output, then it will appear interleaved with all the other applications in the farm.
- **Common keyboard input** : Keyboard input which would be common to all applications is also readable from a file, and the manager distributes this along with the tasks specifications. This avoids any user interaction with the farm once it is running, which ensures the fastest possible through-put. Alternatively, the farm can arrange to read keyboard input whenever a task is to be sent to a free processor.
- **Handling the host server** : In this type of farm, results are "achieved" by writing data in little pieces to files held on the host, rather than a single block result at the end of the task. To keep the farm independent from the application, an OCCAM filter can be used to intercept server access commands and package them into a suitable byte vector before releasing it to the farm. The farm controller

unpacks the vector and forwards commands to the server. A paired decoding filter operates on the input channel to the application.

A further technique used here is to give short-circuit replies to the workers in place of the server. The need for this arises because all server communications involve a reply handshake, which is frequently ignored by the application anyway. The server filter can detect the use of commands to which this technique is appropriate, and immediately acknowledge them. Handling commands like this where possible also reduces the amount of traffic in the rest of the network, further contributing to overall performance.

- **Command-line parameters** : With C applications, the command-line parameter mechanism often employed is still operable. The farm manager takes information from its task specification file and makes them available to the application. Note that the application does not need to be modified at all. For Pascal and FORTRAN applications, it is not possible to read the command line parameters, so the job specification is provided exclusively through keyboard data information sent to the application.

Alternative implementation

An alternative implementation of a farm requiring many workers to access the host, but requiring no user interaction at all, is to simply use server protocol multiplexers as routers. The multiplexers understand a superset of the standard server protocol (so that workers can ask the farm controller for new tasks when they are ready for work). By ensuring that the route that the workers's request packet has to travel in order to reach the farm manager is locked by the multiplexer until the farm manager responds, the system avoids having to make routing decisions at each node if the network is not simply linear (eg, trees, forks, pipes, stars). Although simpler to implement, this type of arrangement will typically give lower performance than the Router/Merger approach.

This approach was used with SPICE in [5] for a simple star network of three worker transputers.

10.7.7 Farming a heterogeneous processor application

Scenario

Consider farming any application which co-executes on another processor type, where all the transputer code is to be farmed. The case of farming part of a mixed-processor system is the same as in Section 10.7.5, because the other processor does not communicate directly with the module to be farmed.

A good starting point for farming a heterogeneous processor application is a working non-farmed system, with the transputer module to be farmed clearly identified.

Implementation

The host source needs to be altered slightly for maximum effectiveness, because results are likely to be returned out of order. Also, using the **HostInterface** source shown earlier, does not guarantee that every job request leads to a valid result. The original access protocol between the host and transputer is modified to accommodate out-of-sequence results and dummy results. The protocol still follows the same pattern of an outgoing followed by an incoming sequence, but the important thing is that the incoming message must identify *what* work packet the result corresponds to, and how how to integrate the results into the host environment.

Often, calling a function or procedure causes variables to be written to which are dependent on the value of the parameters at the time of the function call. For example, a parameter **i** would be passed to a function, and used to reference an array element or scalar variable in the calling environment. Farming will result in out-of-sequence data being returned, which obviously must be used with care in the calling environment. Any writeable variables should be handled by a **ResultsManager** procedure which ensures that the returned results are correctly matched up with the original parameters, and correctly integrated. To achieve this, additional data is sent to the farm and returned with the results.

Incoming work requests to the farm from the host will be not accepted by the farm controller until there is a worker available (therefore the host can be held up at this point). The host keeps track of how many items

the farm is processing, by incrementing a counter each time work is sent to the farm, and decrementing it for every valid reply returned.

For example, consider the farming of the part ported function **HugeTask**, originally part ported in Section 10.6.8. The host stub for **HugeTask** now only handles outgoing messages directly :

```

procedure HugeTask (a, b:integer); { stub }
begin
  outbyte(ValidJob); { intercepted by HostInterface }
  outword(OpCalc1); { Do HugeTask operation }
  ... send any info to slot results into original environment
  outword(a);
  outword(b);

  { originally, HugeTask := inWord;}
  ResultsManager (1);
  { Handles ALL incoming messages }
  { The parameter 1 increments what's in the farm }
end;

```

Originally **HugeTask** was a host function. It has been converted to a host procedure to prevent anything writeable being written to directly following execution of **HugeTask** — this is because the result returned is unlikely to correspond to the *same* set of parameters as were just sent to the farm. All writeable variables must be handled by the **ResultsManager**, unless it does not matter in what order the data is written.

To allow the results to be returned out of sequence, a host **ResultsManager** procedure is used to handle *all* received handshakes from the transputer farm controller :

```

procedure ResultsManager (AddToFarmCount : integer);
var
  ... declarations
begin
  InFarm := InFarm + AddToFarmCount;
  ReplyTag := inbyte; { Valid or not valid results }

  if (ReplyTag = ValidResults) then
  begin
    InFarm := InFarm - 1;
    Command := inWord;
    if (Command = OpCalc1) then begin
      ... receive data from HugeTask operation
      ... handle all order-sensitive writes
    end { OpCalc1 }
    { list other operations here }
  end {ValidResults}
end;

```

To purge the farm of all tasks in it, the host **PurgeFarm** procedure gathers all results together by sending enquiries to the transputer **HostInterface** :

```

procedure PurgeFarm;
begin
  repeat
    outbyte(EnqFarm); { intercepted by HostInterface }
    ResultsManager (0);
  until InFarm <= 0
end;

```

Alternative implementation

By altering the application structure slightly, the complexity of a part port farm described above can be reduced to that of just a part port. This is done by arranging that instead of actions on the host (or other processor)

directly feeding to the **HostInterface**, they instead feed a small stub-like auxiliary routine on the root transputer with a total job specification. By specifying the total amount of work to be done at the outset, the host is freed from the issues of handling out-of-sequence results. The auxiliary routine on the transputer divides up the task into sub-tasks and farms it out to the **FarmInterface**. When all the work is done, the results are packaged up and returned in one consolidated communication to the host. The host part of the application would typically receive an entire array containing all the results in one go.

In this way, the host does not know that farming has occurred, and therefore needs no modification if the original part port was implemented to communicate in this way.

10.7.8 Part port farm example : Second Sight

About Second Sight

Second Sight¹³ 2 is a Turbo Pascal application, with a snazzy front-end using pull-down menus and host graphics. It analyses sets of data and detects trends in them, allowing forecasting future values. The application is suited to farming, because all the sets of data can be operated on independently and concurrently. The code was split into a computational part that executed on a transputer system (doing data modelling), and a user-interface part that executed on the host (unchanged).

All host service accesses are confined to the host computer itself, thereby obviating the need for a standard server. All message passing between the host and the transputer farm was done according to a custom protocol devised for the application. Using techniques outlined in the previous sections, message passing stubs in the host part communicated with the real function bodies on the farm. Certain global data items were broadcast to the transputer farm beforehand the application began in earnest.

For speed of implementation, all the functions that were to run on a transputer were grouped as a single process, and a tagged message protocol was used to indicate which function to run at any time.

The work took about longer than normal (two weeks), because the transputer parts had to be converted from Pascal to OCCAM since at the time (1986), the scientific-language tools were not as powerful as they are now.

Performance

Second Sight running on a single T414-G20 runs about 10 times faster than an 80286/80287 host combination. One T800 transputer runs about 25 times faster. With four T800-G20 transputers, this figure rose to around a factor 100 performance improvement.

10.7.9 Further work

Flood-filling a transputer network

The INMOS development tools are best suited to generating transputer programs for transputer networks of pre-determined size. In other words, prior to run-time, the number and arrangement of transputers is known. Using worm-technology [14], it is possible to write programs that flood-fill a transputer network at boot-time.

The Parallel C and Parallel FORTRAN compilers include utilities for performing a so-called network flood-fill in a homogeneous transputer network. In this way, at boot time all the available processors are utilized depending on their presence, without having to change the software if the hardware arrangement changes. These compilers also support rudimentary but useful farming based on master and worker tasks¹⁴. The application makes explicit calls to run-time library functions which transparently farm the total work load specified by the Master task [9].

¹³Second Sight 2 Copyrights P.H.Todd 1985, and INMOS 1986

¹⁴The worker task cannot access any host facilities.

Extraordinary use of transputer links

It is possible to make a transputer network more resilient to the environment by performing all off-chip communications using the link communication recovery facilities. OCCAM provides a set of procedures which can be used to recover from communications that do not complete successfully for some reason [16, 11]. So, for example, the router and merger processes associated with each farm worker would perform "safe" communications for off-chip channels.

The incorporation of additional communication facilities is very much like adding another layer of abstraction. Again, this layer is independent of the application. The application does not need to be aware of this additional framework, and therefore doesn't require modification.

[16] discusses the use of the facilities provided at the OCCAM level. [11] gives a practical application of using safe link communications in a linear topology farm, showing firstly how a system will continue to operate at reduced performance and with loss of data as transputers begin to fail, and then showing how automatic error recovery (as well as detection) can be achieved.

The Parallel C and Parallel FORTRAN run-time libraries allow direct use of link recovery primitives from C and FORTRAN [9]. This approach involves modification to all applications that want to use this facility.

Overcoming i/o bottlenecks

The nature of farming can impose additional demands on the host services, for example, in terms of much greater access to the host file store. In these situations, there are a few practical steps that can be taken in an attempt to alleviate i/o bottleneck problems.

- Operate the link adapter between the host and the transputer at the maximum speed – 20 Mbit/second communications. This may not assist in disk-bandwidth limited situations, but it will serve to reduce servicing latency.
- Use several link adapters on the host bus, each connecting to a transputer link in the farm. If two link adapters are available, affix one to each end of the farm and stream data through the farm in one direction — in at one end and out at the other. Alternatively, connect all adapters at the farm controller end and use it to increase host to transputer throughput.
- Use a more efficient form of data communication between the host and the transputer. For example, use efficient counted byte vector communications in both directions, coupled with DMA or device driver operation run from the host.
- If disk bandwidth is the real problem, then the above techniques will not significantly result in speedup. The only way to go is for faster disk drives (lower access latency), or use host operating system caching to achieve *apparently* faster disk drives, or ...
- Use several disk drives. For maximum effectiveness, these should each have local busses to avoid performance degradation due to contention clashing. Better still would be for each worker transputer to have exclusive access to a disk drive using the INMOS IMS M212 disk control chip. This can avail data at high speeds directly into transputer links, and is by far the fastest solution. Even if each worker didn't have a whole drive to themselves, the performance increase will still be significant. INMOS has source code available that makes an IMS B005¹⁵ appear as an MS-DOS disk drive [10], allowing easy exchange of files between the bandwidth-limited host drives and the B005 drives *before and after* the farm application executes. An example of this is Steve Ghee's near-real-time pipelined animation machine, which used eight B005 disk drives to store part of a screen's image on each drive.

These techniques can also be used to good effect in increasing communications bandwidth between a transputer farm and non-transputer processor.

¹⁵A b005 is a double-extended Eurocard board which contains a 20 MByte winchester and a 3.5" floppy drive, both under the control of an IMS M212 16-bit transputer.

Comparison between farms and application pipelining

[12] compares the farming concept with that of introducing pipelining into an application. Pipelining is very dependent on the application, has a throughput limited by the slowest element of the pipeline, and is very sensitive to buffering between stages. However, the code required in each stage of a pipeline is generally less than that for a farm worker. Pipelining also accommodates sequential dependencies in an application which would be difficult to deal with in a processor farm.

Pipelining falls outside the scope of this document.

Farms of farms

It is possible to operate a *farm of farms*, whereby an entire application is farmed, but each application consists of a farm of worker sub-tasks. This would be implemented by getting a farm of non-farmed applications working first. The communications protocol between the application and the farm would be carefully documented. Then, the application itself would be decomposed.

Dynamic link switching

In some cases it is important to get data into or out of a farm in a critical time period. In other words, the latency in moving results about must be minimized. It is possible to construct a farm where instead of routing results (or work specification packets) up and down the farm, a direct link connection between the two participating transputers is transiently established for the duration of the exchange. A necklace is still required to permanently connect all transputers, but this is only used for low-bandwidth connection requests and acknowledges. Figure 10.19 shows such a dynamic switched farm, where results are directly routed out of the farm. In the Figure, the work request packets are still distributed conventionally.

This hardware could be constructed using an off-the-shelf B008 or B012 TRAM motherboard, which both contain INMOS C004 cross-bar link switches. A dynamic switched farm similar to this was found to give 15% to 20% better performance in a farmed MandelBrot application¹⁶.

10.8 Planning the structure of a new application

This document has concentrated on taking existing non-OCCaM applications and showing how they can be executed on transputers in a variety of circumstances. Now, a few reminders are called for concerning how one would structure a *new* application so as to make a future transition to the transputer architecture easy.

Here is a summary of some key ideas to guide writing a new application :

- Confine all input / output to a small part of the application. In a transputer implementation, this part should be held within the root module to avoid incurring routing overheads.
- Confine all machine-dependent parts to a small part of the application. This is similar to the above requirement, except that the target-dependent parts may have to stay on the target in a transputer implementation, suggesting some type of part port.
- Structure the application in independent blocks, with well defined interfaces between the blocks. This simplifies implementation as modules. Be aware of the frequency and amount of traffic that would have to pass between these interfaces in a modular scenario.
- Try to arrange that the compute-intensive parts operate on independent data, which facilitates a farm implementation.
- Try to arrange for some computation opportunities during periods of I/O, *especially* those involving user interaction.
- Take into account the underlying hardware booting/reset authorities.

¹⁶Using the Esprit P1085 SuperNode's analogue link cross-bar switching. Research by Sally Baker at RSRE, Malvern.

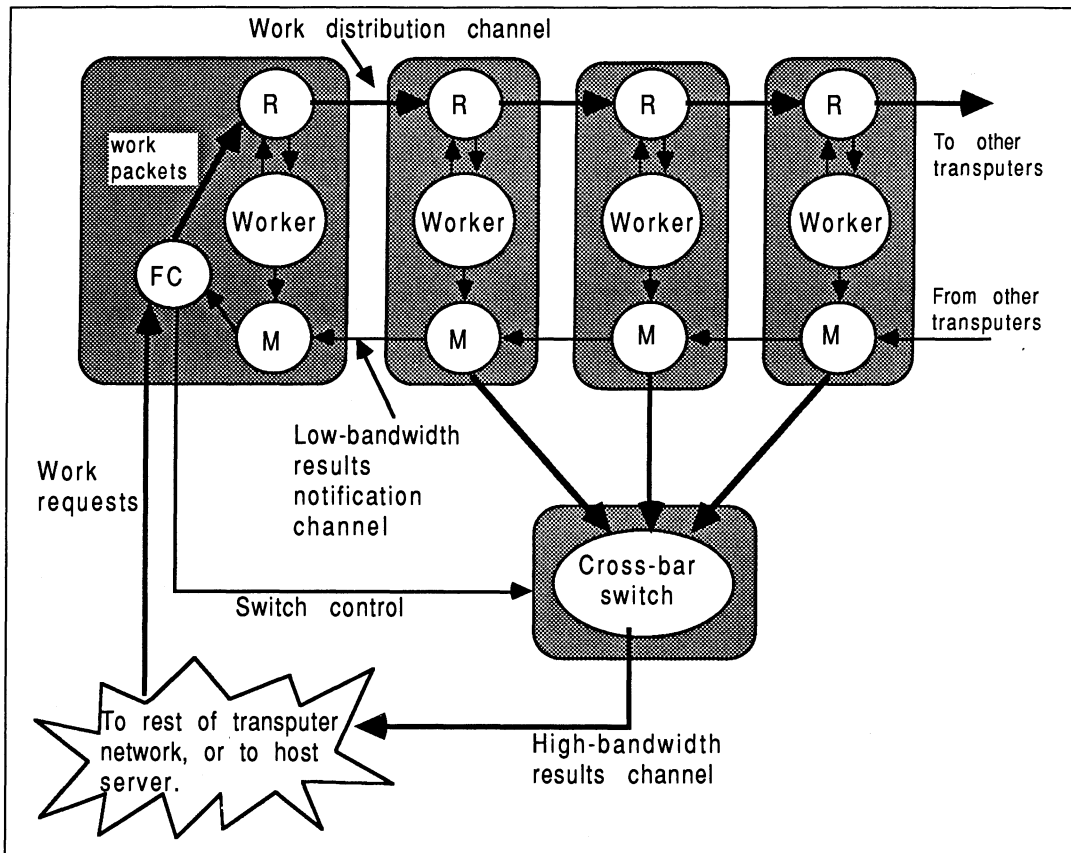


Figure 10.19 A dynamically re-configurable farm

10.9 Summary and Conclusions

Existing applications spanning a wide spectrum of target environments are easily adapted to the transputer. Transputer software is fast, expandable, maintainable, and portable.

The techniques described have all been incremental. Each step builds logically on the operable stages before them. Capabilities and sophistication introduced in this phased way minimize the risks of failure and delay in porting, allowing stratification of the amount of effort in a project.

INMOS provide off-the-shelf slot-in hardware and software components to assist with application porting, parallelization, and farming using transputers. On the hardware side is the TRAM module and motherboard range. On the software side is the farming support, mixed-processor communications support, and the development tools.

Together, these provide an unrivaled facility for quickly putting together a cost-effective system tailored to the exact requirements of the application. The modular nature of both the hardware and the software components allow an implementation to adapt to the changing requirements of an application.

10.10 References

- 1 *The Transputer Databook*, INMOS Limited
- 2 *occam-2 Reference Manual*, INMOS Limited, Prentice Hall 1988, ISBN 0-13-629312-3.
- 3 *Using the occam toolset with non-occam applications*, INMOS Technical Note 55, Andy Hamilton, INMOS Limited, Bristol.
- 4 *INMOS Spectrum*, containing a brief description of the products in INMOS' portfolio.
- 5 *Porting SPICE to the INMOS IMS T800 transputer*, INMOS Technical Note 52, Andy Hamilton and Clive Dyson, INMOS Limited, Bristol.
- 6 *The HELIOS User's Manual*, Perihelion Software Limited.
- 7 *Fundamentals of Operating Systems*, A. M. Lister, University of Queensland, third edition, MacMillan. ISBN 0-333-37097-X, and ISBN 0-333-37098-8 pbk. Refer to Appendix for Monitors.
- 8 *Program design for concurrent systems*, INMOS Technical Note 5, Philip Mattos, INMOS Limited, Bristol.
- 9 *Interface TRAMs*, INMOS Technical Note 42, Phil Atkin, INMOS Limited, Bristol.
- 10 *Using the IMS M212 with the MS-DOS operating system*, INMOS Technical Note 50, Jamie Packer, INMOS Limited, Bristol.
- 11 *Exploiting concurrency; A Ray tracing Example*, INMOS Technical Note 7, Jamie Packer, INMOS Limited, Bristol.
- 12 *Communicating Process Computers*, INMOS Technical Note 22, David May and Roger Shepherd, INMOS Limited, Bristol.
- 13 *Performance Maximization*, INMOS Technical Note 17, Phil Atkin, INMOS Limited, Bristol.
- 14 *Exploring Multiple Transputer Arrays*, INMOS Technical Note 24, Neil Miller, INMOS Limited, Bristol.
- 15 *Parallel C User Guide*, 3L Limited, Scotland.
- 16 *Extraordinary use of transputer links*, INMOS Technical Note 1, Roger Shepherd, INMOS Limited, Bristol.
- 17 *IBM Technical Manual*, Personal Computer AT, March 1984, Document 1502494.
- 18 *Transputer White Pages*, Software and Consultants directory, INMOS Limited, Bristol.



Using the D705B occam toolset with non-occam applications

11 Using the D705B occam toolset with non-occam applications

11.1 Introduction

There is a planet-wide plethora of existing C, Pascal, and FORTRAN software which could benefit from execution on INMOS transputers [1]. Transputers are fast, flexible, and fun. And cost-effective too. Transputers offer an unparalleled opportunity for incrementally upgradable multiple-processor solutions.

In the past, most of the available transputer software support has been centered on the OCCaM [2] programming language, which was developed by INMOS especially for the transputer. Now, development systems for a number of popular languages are available from INMOS and third parties. These development systems can accommodate a range of target and development environments.

This document explains, in programmers terms, how one can use the INMOS development systems to support existing non-OCCaM applications for execution on single or multiple transputers across a variety of hosts. For information concerning the actual *modifications* required to the structure of a non-OCCaM application, in order to fully exploit the parallelism offered by transputers, the reader is directed towards [3].

11.1.1 Article notes

This article places emphasis on the INMOS D705B OCCaM toolset. However, VAX and Sun-3 versions of the OCCaM toolset are available [4]. Everything shown here in relation to the D705B is also applicable to any other development platform. Three dots . . . will be used to represent areas of hidden source text in any language. Hexadecimal numbers will be prefixed by the hash character '#'. A **typewriter** font denotes program text (OCCaM or otherwise). For information on the OCCaM language the reader is advised to refer to [2]. The % symbol is used as a one character wild-card in D705B toolset file names. The term "EOP" represents "Equivalent OCCaM Process". An EOP consists of compiled C, Pascal, or FORTRAN, with the necessary run-time library support, linked together with special OCCaM interface code.

Many thanks to the INMOS Bristol Software Group for their assistance in the preparation of this document.

11.2 Background information

11.2.1 Transputers

The INMOS transputer consists of a high-performance processor, on-chip RAM, and inter-processor links, all on a single chip of silicon. Program variables in on-chip RAM are accessed much faster than if they were off-chip. The inter-processor links are autonomous DMA engines, and permit any number of transputers to be connected together in arbitrary networks. The external memory interface allows linear access to a total memory space of 4 gigabytes.

The T800 and T425 transputers have 4 Kbytes of single-cycle on-chip RAM (40ns access time on a 25 MHz part), and the T414 has 2 Kbytes. The on-chip RAM is usually at least four times faster than the external memory provided with most transputer boards, depending on the hardware design of the board. The fastest external memory supported by the transputer is three-cycle (two cycle on the T801), with most boards using four- or five-cycle memory — using external RAM will *not* make programs run three to five times slower.

For further information on the transputer family, the reader is directed to [1].

11.2.2 The transputer / host development relationship

In the development environment, the transputer is normally employed as an addition to an existing computer, referred to as the host. Through the host, the transputer application can receive the services of a file store, a screen, and a keyboard. This document assumes an IBM PC or compatible host, in so far as it makes reference to some MS-DOS specific features — there are equivalents for the other toolset platforms. For a more thorough guide to product availability, please refer to [4].

The transputer communicates with the host along a single INMOS link. A program, called a server, executes

on the host at the same time as the program on the transputer network is run. All communications between the application running on the transputer and the host services (like screen, keyboard, and filing resources) take the form of messages. The standard transputer C, Pascal, and FORTRAN development systems use a server called **afserver**. The D705B OCCAM toolset, along with the INMOS Parallel C and Parallel FORTRAN development systems, use a server called **iserver**.

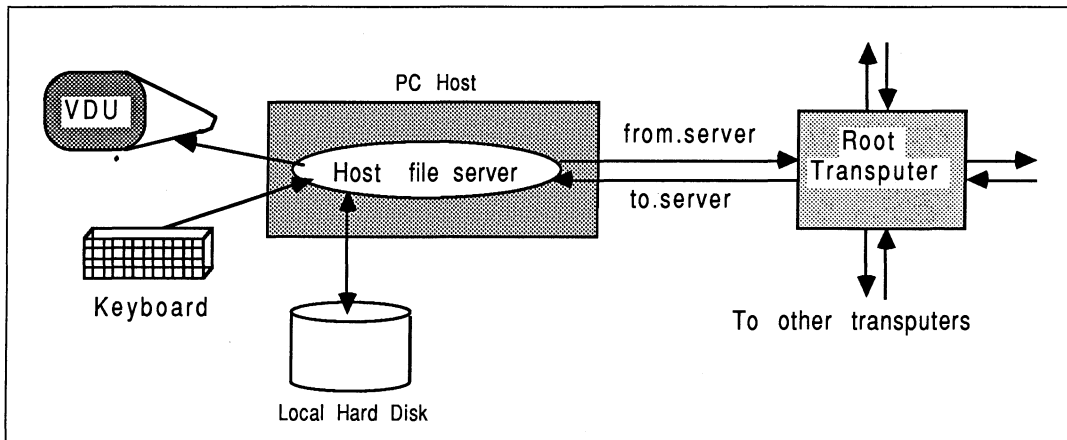


Figure 11.1 The transputer / host development relationship

The *root transputer* in a network is the transputer connecting to the host bus via the link adapter. Any other transputers in the network are connected together using INMOS links, to the root transputer. A transputer network can contain any size and mix of transputer types.

The relationship between the transputer and the host during software development does not impose restrictions on the way the transputer is employed in the target environment.

11.2.3 Connecting transputers together

The INMOS transputer development and evaluation boards use a triplet of signals to control and monitor the status of a transputer network connected to them. These signals are called *reset*, *error*, and *analyze*, and are all used in three ports called *up*, *down*, and *subsystem*. This allows a hierarchy of transputers in a network, where some transputer board can be given the authority to reset and analyze others.

The *down* and *subsystem* ports can assert the *reset* and *analyze* signals to control boards connected to them, and in turn monitor the *error* signal of the sibling board. The *up* port receives the *reset* and *analyze* lines from its parent board, and is used to feed back the status of the *error* line to the parent. On any given board, a connection is made between the *down* or *subsystem* ports to the *up* port on next board. If the *down* port is used, then both boards are at the same hierarchy. If the *subsystem* port is used, then the child board is at a lower level of hierarchy than its parent.

With the OCCAM toolset, a single bootable program is created which contains code for *all* the transputers in the network. The host (PC) computer should have the authority to monitor and control the reset, analyse, and error signals for the whole network. Therefore, when using the toolset software to develop multi-transputer programs, all transputer boards should be connected "down port to up port" from the root transputer outwards. If this is not done, then

- It will be impossible to load the whole network without taking additional steps to ensure that all transputers are correctly reset and analysed.
- The host file server will be unable to monitor the error situation in the network, which will impair the use of the post-mortem debugger.

For users familiar with the INMOS Transputer Development System (TDS), the network attached to the root transputer board is normally connected to the *subsystem* port, rather than the *down* port. This allows the TDS to monitor and control a transputer network, without the risk of itself hanging up due to an execution error in the network. It should be noted however, that this type of connection is not preferred when using the toolsets.

11.2.4 The other OCCAM toolsets

Equivalent versions of the INMOS D705B OCCAM toolset exist for the VAX and Sun-3 environments. These development systems contain the same components and libraries, they accept the same command line arguments and parameters, and offer compatibility at OCCAM source and object binary levels.

This means that OCCAM source, or compiled/linked object code can be freely migrated amongst these development platforms, and compatibility is guaranteed. So, for example, at the time of writing (April 1989), INMOS did not offer VAX and Sun-3 hosted scientific-language compilers. But C Pascal, or FORTRAN source could be compiled with the PC scientific-language compilers, transferred to a different development platform, and integrated with the rest of the application to be ultimately fully portable across the range of OCCAM toolset development platforms.

11.3 The INMOS scientific-language compilers

The INMOS scientific-language compilers can be used to compile and run a non-OCCAM application on a single transputer. They can also be used to build a compilation unit equivalent to an OCCAM process, which can then be incorporated into a complex mixed-language system using the D705B OCCAM toolset (or the Parallel C and Parallel FORTRAN packages).

This section deals only with the capabilities of the scientific-language compilers, and not with those of the D705B OCCAM toolset.

11.3.1 The compilers

In connection with the PC environment, the scientific-language compilers discussed in this document are :

C, Version 1.3	As defined in Kernighan and Ritchie "The C Programming Language", Prentice-hall, 1978. INMOS Part no : IMS D711C
Pascal, Version 1.2	As defined in BS6192:1982, Functionally equivalent to ISO 7185. INMOS Part no : IMS D712C
FORTRAN, Version 1.1	Based on ANSI FORTRAN 77, Defined in ANSI X3.9-1978 with extensions. INMOS Part no : IMS D713C
Parallel C, Version 2.0	As defined in Kernighan and Ritchie "The C Programming Language", Prentice-hall, 1978. INMOS Part no : IMS D711D
Parallel FORTRAN, Version 2.0	Based on ANSI FORTRAN 77, Defined in ANSI X3.9-1978 with extensions. INMOS Part no : IMS D713D

INMOS scientific-language compilers are additionally available for the VAX environment. Remember that binary object code produced by the PC scientific-language development systems can be integrated with the OCCAM toolsets on a different development platform. For details concerning the current product availability and part numbers for the products, refer to [4].

Features

Each scientific-language system offers some useful features over and above those required by the respective standard. The features common to all the scientific-language compilers are listed below :

- They support T414 and T800 transputers. At the time of writing (January 1989), support from INMOS and other manufacturers for the 16-bit transputers (such as the T212/T222 and M212) is in progress.
- In the PC environment, most of the scientific-language tools execute on a transputer board connected to the PC. They can run on any 32-bit transputer. In other environments, such as the VAX, the tools are executed directly by the host computer, but create code for a transputer network.
- The same linker and loader are supplied with C V1.3, Pascal V1.2, and FORTRAN V1.1, for flexibility without requiring additional tools. The D705B occam toolset, D711D Parallel C V2.0, and D713D Parallel FORTRAN V2.0, all use a different but more versatile linker and loader.
- There is a standardized, language-independent, procedural calling interface to access non-occam code.
- 2 Kbytes of the transputer's fast on-chip RAM is reserved for use as a run-time stack.
- Separate compilation program units are permitted in any language.
- One can repeatedly execute the compilers and linker without reloading. This is useful when there are several operations that have to be done consecutively, using the same tool.
- The tools support the host operating system's terminal i/o redirection and piping.
- There are two versions of run-time libraries supplied for each transputer target, depending on whether the application program requires host i/o support.
- The scientific-language run-time library mechanism allows component library modules to be selectively linked.

11.3.2 Using the scientific-language compilers in the simplest case

A single transputer, single non-occam process, is the special simplest case where the occam toolset is not required. It is possible to compile and run a scientific-language process on a single transputer in as few as three commands! These systems are constructed using the pre-compiled binary object files supplied with each of the scientific-language transputer compilers, using a command structure which is similar for C, Pascal and FORTRAN applications. A transputer bootable file is one which contains enough information to allow it to be sent to a transputer (network) by the host file server, and executed. A bootable file is created by linking the compiler's object output with various run-time support components, and prepending a bootstrap loader.

Each command shown below causes the appropriate tool to be loaded onto the transputer board, and run with the appropriate parameters. All the compilers accept their respective source-level input, and produce by default a binary object file as output. The linking command causes the compiled binary object file to be linked with the appropriate run-time library, and also with a supporting fragment of occam which is known as the "harness". The purpose and content of the harness is described in Section 11.5.

Note that the file name extensions are optional, but are included here explicitly. The filename conventions for the PC environment for binary object files is **.bin**. The scientific-language compilers can optionally produce hexadecimal object code, identified by a **.hex** filename extension. A **.b4** extension identifies a transputer bootable file for a single transputer. Source files for C, Pascal, and FORTRAN have the default extensions of **.c**, **.pas**, and **.f77** respectively.

Building a simple C program

Standard tool operation is :

Operation	T414 target	T800 target
Compile	t4c prog.c	t8c prog.c
Link	t4clink prog.bin	t8clink prog.bin
Run	run prog.b4	run prog.b4

Building a simple Pascal program

Standard tool operation is :

Operation	T414 target	T800 target
Compile	t4p prog.pas	t8p prog.pas
Link	t4plink prog.bin	t8plink prog.bin
Run	run prog.b4	run prog.b4

Building a simple FORTRAN program

Standard tool operation is :

Operation	T414 target	T800 target
Compile	t4f prog.f77	t8f prog.f77
Link	t4flink prog.bin	t8flink prog.bin
Run	run prog.b4	run prog.b4

11.3.3 Loading the tools

Although the user may not be aware of it, all tools are loaded by calling the host file server. This is **afserver** or **iserver** depending on the development system. For systems using the **afserver**, the server is supplied with the name and parameters of the tool to be loaded. For example, the command **t4c world**, to compile the C program **world.c**, is actually doing something like this :

```
afserver -:b \tc1v3\tc.b4 world /t4 -:o 1
```

The **-:b** command is the server's boot command, and causes the file referenced to be sent to the transputer board and executed. The **-:o 1** is concerned with the workspace allocation that the compiler will use on the transputer board. This is an example of using the run-time workspace specification capability described in Section 11.3.8.

The same approach is used for the other scientific-language compilers, and for the linker. For example, the command **t4clink world** does the following :

```
linkt world.bin+\tc1v3\crt1t4.bin+\tc1v3\t4harn.bin,world.b4
```

The plus signs above represent the concatenation of the input files, and the comma separates the list of input files from the output file. The reference to **linkt** calls the **afserver** with the **linkt.b4** transputer bootable linker. This adds the necessary parts from the T414 C runtime library **crt1t4.bin**, and the supporting harness **t4harn.bin**, to make a bootable file called **world.b4**.

For the Parallel C and Parallel FORTRAN compilers, which use the **iserver**, the principle is the same as above, but the boot files and server options are different.

11.3.4 Re-running the tools without reloading them

It is straight forward to re-run the compiler and linker tools described above, without having to boot the tool onto the transputer board each time the tool is used. This is achieved by calling the **afserver** program

directly, but without specifying the boot command (`-:b filename`).

As an example of this, suppose that the C compiler has been loaded onto the transputer board, and set to compile a file called `c1.c` for the T800, using the following command :

```
t8c c1
```

Then to compile separate applications `c2` and `c3` for the T414 and `c4` to `c7` for the T800, but without reloading the C compiler each time, one can use the following commands :

```
afserver c2 /t4 -:o 1
afserver c3 /t4 -:o 1
afserver c4 /t8 -:o 1
afserver c5 /t8 -:o 1
afserver c6 /t8 -:o 1
afserver c7 /t8 -:o 1
```

Note that once a compiler has been loaded, then each time it is re-run, the `afserver` must be given a `-:o 1` directive. This is so that when the compiler is running, it is given the maximum available memory on the transputer board for its own workspace requirements (see Section 11.3.8). For example, to compile the following three FORTRAN programs, use this technique :

```
t8f f1.f77
afserver f2.f77 /t8 -:o 1
afserver f7.f77 /t4 -:o 1
```

The first command here will actually load the FORTRAN compiler, and the remaining two will correctly re-run it for the different processor targets.

The same technique can be used to re-run the linker, and also applies to `iserver` tools.

11.3.5 Running transputer bootable files as MS-DOS commands

It is possible to run any transputer executable `.b4` file as if it were an MS-DOS command. This is done using the `linkt.exe` program supplied with all the scientific-language compilation systems. Make a copy of the `linkt.exe` program but give it the same root filename as the bootable `.b4` program you wish to run as an MS-DOS command; keep the `.exe` extension.

The `linkt.exe` program works by taking the command verb from its command line, adding the `.b4` extension, and calling the host file server `afserver` to load that file from the same directory as the `linkt.exe` was loaded from. When invoking a `.b4` file in this way, the `afserver` is passed the `-:o 1` directive automatically to give the application (if it uses the standard OCCAM harness) one large combined workspace. It is still possible to specify the `-:o 0` directive on the command line to over-ride this, ensuring the run-time stack is placed in on-chip RAM.

11.3.6 The run-time libraries

Each scientific-language comes supplied with two different run-time libraries. This is important when one is developing multiple-process systems. A process which expects to communicate with the host file server must be linked with the full run-time library. A process which uses only the channel communication primitives discussed in Section 11.3.9, plus other functions that do not require to access the host i/o facilities, can be linked with the reduced (stand alone) run-time library. This offers certain advantages in terms of code size, execution speed, and "portability" within a multi-process system.

Each run-time library consists of separately compiled program modules. The full and stand alone libraries have many modules in common — the stand alone library being essentially a subset of the full run-time library. The languages of implementation of the modules include C, IMP, and OCCAM. The library management facilities offered by the linker permit the binary object files produced from different language compilers to be mixed together and referenced as a single entity; the library. Only those library modules that satisfy outstanding external references will be linked into an application by the linker.

At start-up, all the static workspace in the referenced modules in the run-time library is relocated from the non-OCCAM code area to the heap workspace area. This is done because the code area could be in read-only store such as EPROM, whereas the heap workspace must be writeable. The existence of this static data in some component modules prevents the run-time libraries (as a whole) from sharing the re-entrancy property that OCCAM libraries possess.

The component object modules which were used to build each library are also supplied with each scientific-language system, along with control files to allow the linker to reconstruct these libraries. This allows users to create their own libraries, add their own modules to them, and delete unused modules, to suit specific project requirements.

11.3.7 Transputer memory allocation

This section discusses the memory allocation policy used by the scientific-language compilers. An overview of the OCCAM memory allocation strategy is given first, because all scientific-language memory allocations conform to this framework.

The OCCAM memory allocation map

The transputer employs a signed memory address space, which for 32-bit machines begins at **MOSTNEG INT (Mint)** #80000000 and extends up through zero to the positive address space and onwards to **MOSTPOS INT** #7FFFFFFF. External memory is usually decoded at very negative addresses, because in this way it forms a seamlessly-joined contiguous block with the transputer's on-chip RAM. Memory in a system is allocated from the most negative addresses onwards. This is shown in Figure 11.2.

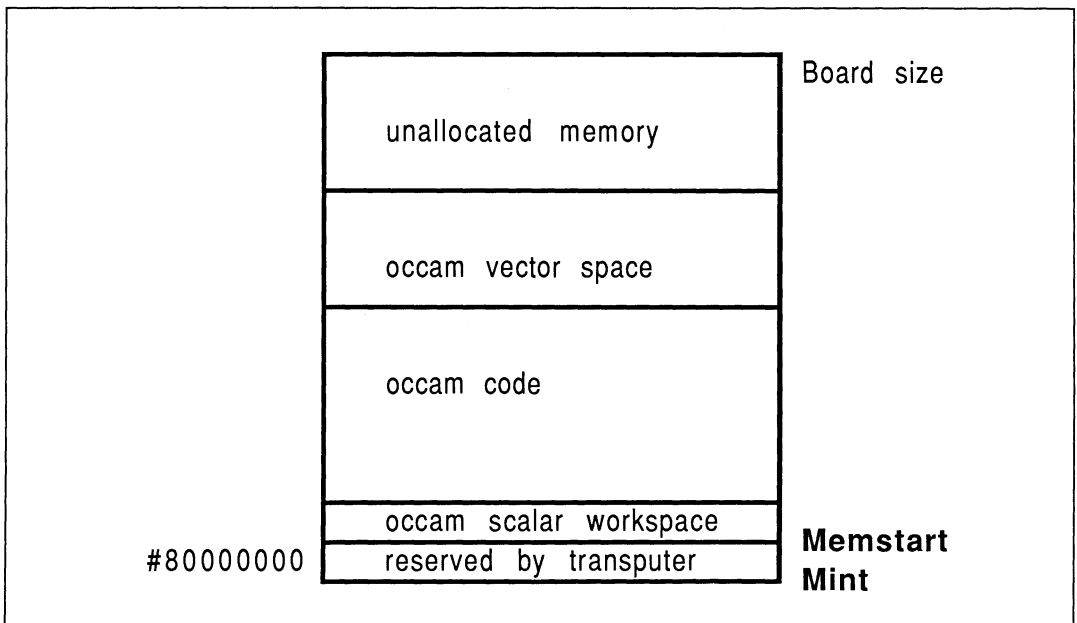


Figure 11.2 The transputer memory map

With reference to the Figure, there are five memory zones in the memory map. Starting at the bottom of memory is an area reserved by the transputer. The first memory location in the transputer not required by the transputer itself is called **Memstart**. On a T414, this corresponds to address #80000048, and on the T425 / T800 series corresponds to #80000070. The host file server loads the boot file, using memory from **Memstart** onwards.

The Figure shows that scalar OCCAM workspace is placed as low down in memory as possible, starting in on-chip RAM just above **Memstart**. The OCCAM compiler places the most recently declared variables in the lowest workspace slots.

Directly following the scalar OCCAM workspace is the code area. This represents the concatenation of all the object files comprising the application, plus any library routines that were referenced. If any of the OCCAM source was compiled with separate vector space on, then after the code area follows the vector space area. Above this, the memory on a transputer system is unallocated.

This memory arrangement is made possible because, in OCCAM, all data allocation is static. This means that after compilation and linking, the loader knows exactly the data requirements of the program, for both scalar and vector workspaces.

After the boot file has been loaded by the file server, the bootstrap code does a **KERNEL.RUN** of the process code, and execution on that processor begins.

All memory allocation in the scientific-language systems is ultimately under the control of some standard OCCAM specification. All memory allocation in the scientific-language systems conforms to the OCCAM memory allocation policy described above. This fact should guide one's understanding of the memory allocation diagrams in Section 11.5.4.

The scientific-language memory allocation map

Memory for scientific-language workspace usage is allocated from an integer vector representing all the available memory left on the board once the application has been loaded. This vector extends from the top of the board memory right down to the top of the OCCAM vector space zone. This memory area is shown in Figure 11.2 as unallocated memory.

Using only the tools provided with a scientific-language compiler, a single transputer single process system can be created¹. The memory allocation in this system is shown in Figure 11.3. This represents the memory map of the standard OCCAM harness supplied with each scientific-language system (for creating a single process single processor system).

All the scientific-language compilers operate with two logical workspaces : a run-time stack and a combined heap and static data area. Depending on a run-time option, and various decisions made when compiling the OCCAM support software, the physical realization of these logical workspaces varies.

Figure 11.3 shows this reserved run-time stack area in the OCCAM scalar workspace zone. On a T414 transputer, this uses up all the on-chip RAM. Even if the user does not run the application to make use of this stack, this memory is always reserved when using the standard OCCAM harness. The Figure also shows a run-time stack at the top of the memory map, and a heap lower down. Only one stack area is ever used by a scientific-language process at any one time.

11.3.8 Implementation details

These features are common to all the scientific-language compilers. Some are designed to allow good use of the transputers' on-chip RAM. Others simplify the accommodation of changing development situations.

The run-time stack

The run-time stack is known as a "falling" stack. The stack pointer starts off high in memory and descends as space is allocated. Called functions will have their workspaces placed at lower addresses than the caller. The loader will attempt to determine the size of the target board, so it can make best use of the available memory by placing the top of the stack at the very top of physical memory.

If the user elects to use the on-chip stack (assuming it is sufficiently spacious for the application), then the space at the top of memory will not be used. If the off-chip stack is selected for use, then it is important that as the stack grows downwards and the heap grows upwards, "never the twain shall meet". Heap allocation

¹The Parallel C and Parallel FORTRAN systems additionally support multiple transputers.

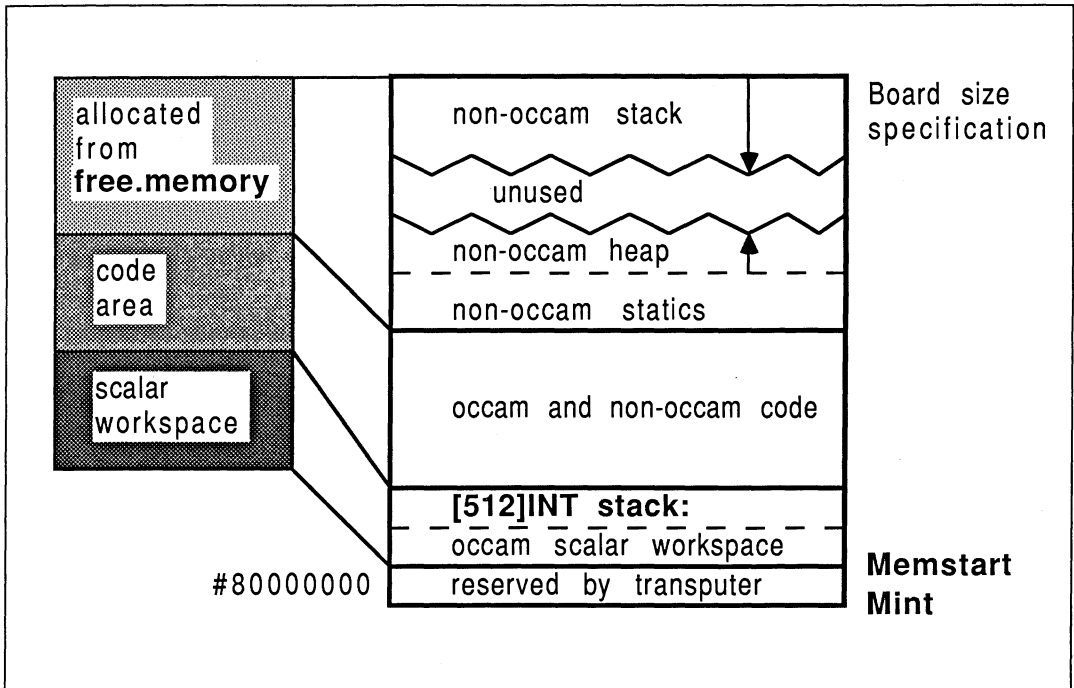


Figure 11.3 The scientific-language compiler memory map

requests are range checked to ensure that the stack is not about to be overwritten — but for performance reasons, this is not true of stack allocation requests. The stack *can* overwrite the heap area, but not the other way round. If any workspace overwriting occurs, the program will fail in unpredictable ways.

The run-time heap

The run-time heap is known as a “rising” heap. This means that it starts off at a low memory location and uses successively higher memory locations as data is added to it. The heap directly follows from the static data storage area. The heap is used typically for variable-length memory allocations, for items such as strings, arrays, and the dynamic commands like `malloc()`. Compared to the stack, allocation requests for heap space are much more infrequent, and tend to be for larger data items. This means that there is a comparatively low overhead in checking run-time requests for heap space, to ensure that the heap is not about to overwrite the stack.

Section 11.5.4 discusses ways of calculating and fine-tuning the amount of stack space and heap space to reserve for non-OCCAM processes in multiple-process systems.

Selecting the run-time stack

The user can select to use the run-time stack either in on-chip RAM or in external memory.

If the *whole* of the stack for a program can be accommodated within 2 Kbytes, then the on-chip stack can be used on either the T414 or the T800. In this case, only the heap and static data area is placed in external memory — the default assumed by the standard harness implementation. The standard harness reserves an on-chip stack regardless of whether it is used.

If the size of the stack is expected to be larger than 2 Kbytes, then the off-chip stack area is used, and the application will therefore have all its workspace off-chip. The parameter `-:o 1`, supplied to the `afserver`

at run-time, specifies that all workspace is to go off-chip. Note that no action is required at compile-time or link-time to specify the location of the run-time stack. This facility should be used while developing a program, for which one is uncertain of the requirements in terms of stack size. Refer to Section 11.5.4 for details on dynamic fine-tuning of workspace requirements.

Note that the Parallel C and Parallel FORTRAN development systems operate slightly differently than described above. With these systems, the "standard harness" does not reserve an on-chip stack area unless this is specified when the bootstrap is prepended. In this way, no on-chip RAM is wasted needlessly. Using an option on the bootstrap tool, the programmer specifies the size of a separate stack (if one is required), and this is placed as low down in memory as possible.

Placement of the code

Some on-chip RAM can normally be used for code storage. On the T414, using the **afserver**-based development systems, there is no internal RAM available for code storage. The **iserver**-based tools, because they don't reserve unused stack space, *do* permit code storage on-chip in a T414. The T800/T425 families have at least 2 Kbytes of on-chip RAM that is not reserved for the variable stack, available as a code store. The **iserver** tools avail even more.

The ordering of the files to link is critical for the performance of the program, because code placement on the processor is determined by the linking order of the binary object files. Programs will therefore run faster if small, speed-critical routines are placed at the beginning of the list of files to be linked, and the OCCAM calling process is placed at the end.

It is not possible to have the whole of on-chip memory on the T800 exclusively as a stack or code area. It is also not possible to have part of the stack on-chip and part of it off-chip. This is due to the implementation of the development tools.

These restrictions on the specification of the scientific-language compilers were adopted for the following reasons. Studies showed that in the event of a trade-off in the use of on-chip memory between code and data, it is generally more efficient to permit some data to be placed on-chip (in the stack) rather than only having application code on-chip. This is due to the high density of transputer machine code, and the transputer's hardware instruction pre-fetch mechanism. Therefore, any transputer can offer some on-chip RAM for stack purposes, but the availability of on-chip RAM for code depends on the transputer and the family of development tools.

The static data area

Physically, the initialized static data area is placed at the bottom of the heap workspace area. This is placed immediately above the mixed-object code area. The size of the initialized static area can be determined at compile-time, and all the compilers generate a pre-initialized "image" of this static data, rather than generating code to perform a run-time initialization of this area. Two draw-backs of the adopted method are that large static initialized arrays result in large binary object files, since the value of each element appears explicitly. However, in addition to this, some run-time initialization is performed by using embedded initialization information in the code output by the compiler for each module (some items cannot be initialized at compilation or linkage phases). Each static data variable has initialization data embedded in this way; a byte of initialization data for every byte of static data required by the variable.

The run-time initialization involves relocating the static data from the code area to the static/heap workspace area, and initializing it prior to execution. This is because the code area could be in read-only store.

The scientific-language process communications interface

The scientific-language systems create compilation units which can be made into an equivalent OCCAM process (EOP). The interface to this compilation unit was devised for flexibility, and is not suitable for direct inclusion into a parallel system — it should always be wrapped in a layer of OCCAM, described in Section 11.5.

The "raw" communications interface to an EOP takes the form of two arrays of pointers to channels. These are passed as arguments to the process by the surrounding OCCAM environment, and consist of one array of pointers to input channels, and one array of pointers to output channels. The run-time libraries for the

language involved provide access to these channels. The general interface to an EOP is shown in Figure 11.4.

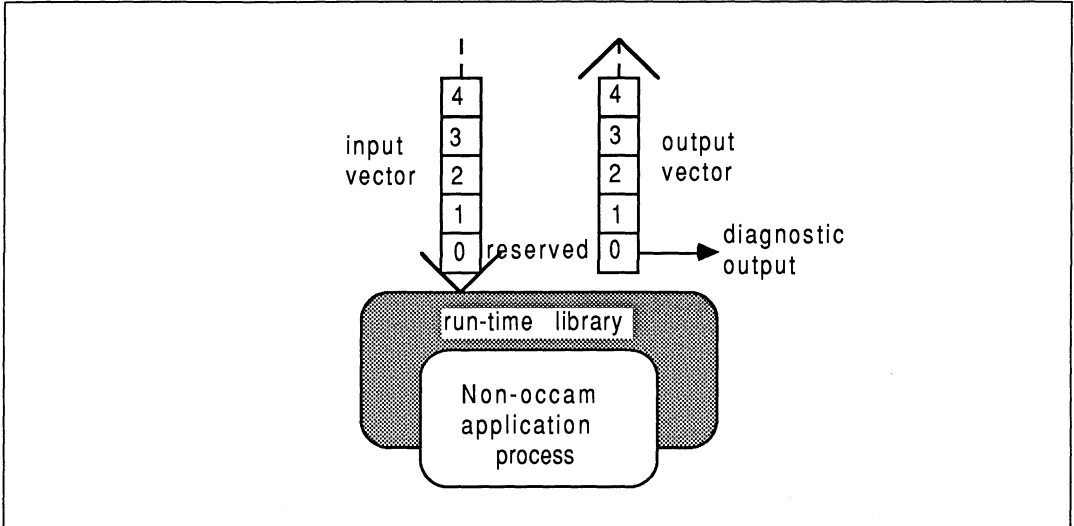


Figure 11.4 General scientific-process interface

Depending on the run-time library used with a particular scientific-language process, some elements of the channel address vector will be reserved :

- If the EOP uses the full run-time library, then the first two elements of both vectors are reserved. Element 0 of the output vector is used for run-time library diagnostic output, and element 1 of both vectors carries host i/o traffic as defined by the language's input / output facilities.
- If a C or FORTRAN EOP uses the standalone run-time libraries, then only element zero of both vectors is reserved.
- If a Pascal EOP uses the standalone run-time library, then no elements are reserved.

Either vector of pointers to channels can be arbitrarily large, and the user is free to use them for interconnection to other processes, OCCAM or otherwise. In general, elements 0 and 1 of the input and output channel pointer vectors should never be used by the programmer; only elements 2 and upwards should be used. Section 11.5 shows how best to conceal the implementation interface to non-OCCAM components in a system, using the D705B occam toolset.

11.3.9 Scientific-language channel i/o support

In OCCAM, parts of an application communicate by sending messages to each other on channels. This is also true of the scientific-language implementations. Channels provide unbuffered, unidirectional, synchronized, point-to-point communications between two concurrent processes. Each scientific language is provided with four message-passing facilities by means of run-time library functions, which map directly onto the transputer's channel i/o instructions [5]. These facilities in each scientific-language behave exactly the same as OCCAM's input (?) and output (!) primitives, and are outlined below :

C support

The four channel communications functions for V1.3 C are as follows :

Command	Parameters	Description
<code>_outword</code>	<code>w, chanp</code>	word output
<code>_outbyte</code>	<code>b, chanp</code>	byte output
<code>_inmess</code>	<code>chanp, buffer, nbytes</code>	message input
<code>_outmess</code>	<code>chanp, buffer, nbytes</code>	message output

The parameter types in the above table are as follows :

```
int w, nbytes ;
CHAN *chanp;
char b;
char buffer[];
```

The C `main()` body is given the following arguments :

```
typedef int CHAN;
main(argc, argv, envp, in, inlen, out, outlen)
int argc, inlen, outlen;
char *argv[], *envp[];
CHAN *in[], *out[];
```

Elements of the vectors `in[]` and `out[]` correspond exactly to those described in the previous section about the scientific-language program interface.

The channel communication primitives shown above are made available by including this header file in all compilation units that perform message passing :

```
#include <chanio.h>
```

These examples assume that the messaging routines are called from within the `main()` function body, otherwise the `in` and `out` vectors declared as arguments to `main()` are not in scope :

- Receive on channel 3 a one byte value and store as an integer

```
int tag=0;
_inmess(in[3], &tag, 1);
```

Notice that the `tag` is initialized to zero before the byte read. This is because only the least significant byte of the integer will be affected by the byte read, so it is advisable to initialize the whole integer to a known and sensible value before operating on only part of it.

- Receive on channel 2 a 4 byte integer, then display it

```
int value;
_inmess(in[2], &value, 4);
printf("%d\n", value);
```

- Receive on channel 4 a `double`, then send it out on channel 3

```
double item;
_inmess (in[4] , &item, 8);
_outmess(out[3], &item, 8);
```

- Output a byte tag #02 on channel 4, then output integer 3

```
_outbyte(2, out[4]);
_outword(3, out[4]);
```

It is particularly important to notice that in the case of the `_inmess` and `_outmess` functions, the second parameter is the *address* of a buffer containing the actual data. If one uses the `_outmess` to send a word or a byte, be sure not to place a literal constant (ie, a number like 42) as the data. This should only be

attempted with the `_outbyte` or `_outword` functions.

To be able to use the messaging facilities from functions `outwith main()`, and yet avoid passing in the channel pointers as function parameters each time, it is necessary to declare outside `main()` two pointers to these channel vectors. One way of doing this would be as follows :

```

typedef int CHAN;

CHAN **in, **out;          /** This does the scoping **/

main(argc, argv, envp, topin, inlen, topout, outlen)
int argc, inlen, outlen;
char *argv[], *envp[];
CHAN *topin[], *topout[];
{
    ... usual declarations
    in = topin;
    out = topout;
}

```

Only now is it possible to globally reference elements of `in` and `out` from any functions other than `main()`. This is particularly important, because the system may appear to behave as if the channels *were* correctly connected, yet produce incorrect results and fail to terminate if this channel scoping is not correct.

Pascal support

The four channel communications procedures for V1.2 Pascal are as follows :

Command	Parameters	Description
<code>outword</code>	<code>w, channel</code>	word output
<code>outbyte</code>	<code>b, channel</code>	byte output
<code>inmess</code>	<code>channel, buffer, nbytes</code>	message input
<code>outmess</code>	<code>channel, buffer, nbytes</code>	message output

The parameter declarations in the table above are as follows :

```

w, channel:INTEGER;
b: CHAR;
VAR buffer:UNIV CHAR;
nbytes:INTEGER;

```

These are made available by including the following file with one's application code, and compiling the application with the `/x` option (which has the effect of allowing certain extensions to the ISO 7185 / BS6192:1982 Pascal definition to which the compiler normally conforms) :

```

$include '\tp1v2\channels.inc'

```

The directory `tp1v2` is the home directory for the version 1.2 Pascal compiler, so it is specified in the path for the include file.

The `UNIV` type of parameter, shown above in procedures `inmess` and `outmess`, provides a loophole for breaking Pascals' strict type checking rules when passing parameters. As an extension to the ISO/BS standards, the reserved word `UNIV` can be prefixed to the type of a `VAR` parameter. This allows the parameter to be specified as a variable of any type.

The channel numbers used with these message-passing procedures corresponds exactly to those described in the previous section about the scientific-language program interface.

Some examples of Pascal channel communications in action :

- Receive a byte called tag on channel 2

```
inmess(2, tag, 1)
```

- Receive an integer called data on channel 3

```
inmess(3, data, 4)
```

- Output an integer called count on channel 2

```
outword(count, 2)
```

- Output a byte #05 on channel 3

```
outbyte(chr(5), 3)
```

FORTRAN support

The four channel communications subroutines for V1.1 FORTRAN are as follows :

Command	Parameters	Description
CHANOUTWORD	VALUE, ICHANNEL	word output
CHANOUTBYTE	VALUE, ICHANNEL	byte output
CHANINMESSAGE	ICHANNEL, BUFFER, NBYTES	message input
CHANOUTMESSAGE	ICHANNEL, BUFFER, NBYTES	message output

The parameter declarations in the table above are as follows :

```
INTEGER ICHANNEL, NBYTES, VALUE
```

Any FORTRAN object — **BUFFER**

It is not necessary to specify any additional information in the source text of your application (as is the case with C and Pascal) before these can be used. They are made available at link-time from the FORTRAN run-time libraries.

The **ICHANNEL** number used with these message-passing subroutines corresponds exactly to those described in the previous section about the scientific-language program interface.

Now, some examples of FORTRAN channel communications :

- Send a real number on output channel 2

```
C      REAL*4 A
      Note that A IS 4 bytes in size
      CALL CHANOUTMESSAGE(2, A, 4)
```

- Receive an integer number from input channel 2

```
C      INTEGER*4 B
      Note that B IS 4 bytes in size
      CALL CHANINMESSAGE(2, B, 4)
```

- Receive into channel 2 as an integer a byte tag (length 1)

```
INTEGER TAG
TAG = 0
CALL CHANINMESSAGE(2, TAG, 1)
```

The **TAG** integer is initialized to zero before reading in data to its least significant byte — the byte read will *not* affect the top 3 bytes in the integer, so to allow direct comparisons in this way it is

sensible to pre-initialize the whole word to a known value.

- Output a byte of value #01, then a word VALUE, on channel 2

```

INTEGER VALUE
VALUE = 1
CALL CHANOUTBYTE (1, 2)
CALL CHANOUTWORD (VALUE, 2)

```

It is particularly important to notice that in the case of the **CHANINMESSAGE** and **CHANOUTMESSAGE** subroutines, the second parameter is the *address* of a buffer containing the actual data. So ensure you never attempt to use literal constants for this parameter. For example, **CHANOUTMESSAGE (2, 0, 1)** will *not* send a byte of value 0 on channel 2 — it will attempt to decode memory at hardware address 0 and send that as a byte. Since positive address space is rarely decoded as physical memory on current production transputer boards, this is certainly wrong and could be dangerous!

Parallel C support

Parallel C version 2.0 offers some additional message passing primitives compared to the C version 1.3. One gains access to these by inserting **#include <chan.h>** in the source.

Command	Parameters	Description
chan_in_byte	in_b, chanp	byte input
chan_in_byte_t	in_b, chanp, timeout	timeout / byte input
chan_init	chanp	initialize a channel word
chan_in_message	nbytes, buf, chanp	message input
chan_in_message_t	nbytes, buf, chanp, timeout	timeout / message input
chan_in_word	in_w, chanp	word input
chan_in_word_t	in_w, chanp, timeout	timeout / word input
chan_out_byte	out_b, chanp	byte output
chan_out_byte_t	out_b, chanp, timeout	timeout / byte output
chan_out_message	nbytes, buf, chanp	message output
chan_out_message_t	nbytes, buf, chanp, timeout	timeout / message output
chan_out_word	out_w, chanp	word input
chan_out_word_t	out_w, chanp, timeout	timeout / word input
chan_reset	chanp	reset channel word

The parameter types in the above table are as follows :

```

char *in_b, out_b;
int *in_w, out_w;
char *buf;
int *chanp;
int timeout;

```

For compatibility reasons, the channel messaging routines supplied with the version 1.3 C compiler are also included, and can be accessed by referencing header file **#include <chanio.h>**.

Parallel FORTRAN support

Parallel FORTRAN version 2.0 again offers a superset of message passing primitives compared to the FORTRAN version 1.1. One gains access to these by inserting **INCLUDE 'CHAN.INC'** in the source.

Command	Parameters	Description
F77_CHAN_ADDRESS	CHANWORD	address of channel word
F77_CHAN_IN_BYTE	IBUFF, ICHANADDR	byte input
F77_CHAN_IN_BYTE_T	IBUFF, ICHANADDR, TIMEOUT	timeout / byte input
F77_CHAN_INIT	ICHANADDR	initialize a channel word
F77_CHAN_IN_MESSAGE	LENGTH, BUFF, ICHANADDR	message input
F77_CHAN_IN_MESSAGE_T	LENGTH, BUFF, ICHANADDR, TIMEOUT	timeout / message input
F77_CHAN_IN_PORT	PORTNO	value of input port binding
F77_CHAN_IN_PORTS	—	number of input ports
F77_CHAN_IN_WORD	WORD, ICHANADDR	word input
F77_CHAN_IN_WORD_T	WORD, ICHANADDR, TIMEOUT	timeout / word input
F77_CHAN_OUT_BYTE	IVAL, ICHANADDR	byte output
F77_CHAN_OUT_BYTE_T	IVAL, ICHANADDR, TIMEOUT	timeout / byte output
F77_CHAN_OUT_MESSAGE	LENGTH, BUFF, ICHANADDR	message output
F77_CHAN_OUT_MESSAGE_T	LENGTH, BUFF, ICHANADDR, TIMEOUT	timeout / message output
F77_CHAN_OUT_PORT	PORTNO	value of output port binding
F77_CHAN_OUT_PORTS	—	number of output ports
F77_CHAN_OUT_WORD	WORD, ICHANADDR	word input
F77_CHAN_OUT_WORD_T	WORD, ICHANADDR, TIMEOUT	timeout / word input
F77_CHAN_RESET	ICHANADDR	reset channel word

The parameter types in the above table are as follows :

INTEGER CHANWORD

INTEGER IBUFF, ICHANADDR, TIMEOUT

INTEGER PORTNO, IVAL

INTEGER NCHAN, ICHANADDRARRAY (NCHAN)

Any FORTRAN object — **BUFF**

Any 4 byte FORTRAN object — **WORD**

For compatibility reasons, the channel messaging routines supplied with the version 1.1 FORTRAN compiler are also available.

11.3.10 Additional support from Parallel C and Parallel FORTRAN

The Parallel C and Parallel FORTRAN compilers have some additional capabilities to support the generation of parallel processes, and also replace the toolset's OCCAM configuration stage with a C-like meta-language.

Parallel C has the concept of parallel threads of execution. A C task can contain several parallel execution threads. All of a task's threads share the same **static**, **extern**, and heap data, and therefore run on the same processor as the governing task. Each thread has its own stack for **auto** variables, which is allocated from the heap of the main task by using a **thread_create** function. A semaphore mechanism is provided to ensure mutual thread exclusion from critical shared data areas. Threads can also communicate with each other by using channels.

Parallel FORTRAN also has a multiple thread facility, but this is more restricted than in Parallel C because FORTRAN sub-programs are not re-entrant — a sub-program cannot call itself, directly or otherwise.

Using threads without due care in synchronizing access to shared data areas with semaphores can introduce errors which are very difficult to pin-point. In contrast to a thread, a task is a more substantial entity. Tasks correspond to the compilation units of the other compilers. Tasks communicate with each other only by using channels. Each task has its own code and data areas which are separate from those of all other tasks.

The Parallel C and Parallel FORTRAN configuration meta-language allows one to specify a process to processor mapping without recourse to an OCCAM specification. The hardware topology is described in terms of **processor** and **wire** statements, which include the host PC as a processor. Each task in the network is identified with a **task** specification which names the task and identifies the number of input and output channels, plus specific requirements such as heap space. Tasks are allocated to processors with the **place** directive, and are interconnected using **connect** statements.

One attraction of the Parallel C and Parallel FORTRAN compilers over the OCCAM toolset software is the flood-filling configurator. This allows applications written in a particular way (a single controller task with arbitrary numbers of identical workers) to be broadcast in a transputer network to automatically take advantage of how ever many transputers happen to be present.

The Parallel C compiler is supplied with a decoder utility which can examine the binary object output from the compiler. It produces a listing showing the source code and the corresponding disassembled machine code. It can also be used on the object output of the V1.3 C, V1.2 Pascal, and Parallel FORTRAN compilers. Note that the utility cannot be used on bootable **.b4** files. The utility is similar to the D705B toolset's **ilist** utility.

For further information on INMOS Parallel C or Parallel FORTRAN, refer to [6,7].

11.3.11 Transputer assembler inserts

The two C compilers described earlier both support the inclusion of transputer assembler inserts. This is not documented for the version 1.3 C compiler because the implementation provided in this case is limited and can give incorrect code generation without notification (for example, if one attempts to access local **auto** variables symbolically). Note clearly that this facility is *not* supported by INMOS. The Parallel C version 2.0 offers a more flexible and correct assembler insert capability.

Usage of assembler

The use of transputer assembler should be restricted to either increasing the performance of short sections of time-critical code, or for direct manipulation of the hardware. The assembler capability in the C compilers is suitable for these tasks, but should not be seen as a means of writing large sections of code in assembler (for this a proper symbolic macro-assembler is advised). And don't try it unless you have access to [5].

A transputer assembler insert is introduced with the **asm** directive. Instruction mnemonics are expressed in lower case. An example of using transputer assembler is shown below :

```
int loc (a)
int *a;
{
    asm
        { ldl 2 ; }
}
```

This function was used in a large FORTRAN application [8] to return the address of a variable passed as a parameter to it. As FORTRAN passes parameters by reference anyway, it is simply necessary to load the parameter into the transputer's A register and return. To understand why the parameter is referenced with a **ldl 2** instruction, the following discussion on workspace allocation is helpful.

Local workspace allocation

Assuming that no temporary variables are required, the transputer C compilers allocate local function workspace as follows :

- Local **auto** variables are allocated from workspace slot 0 upwards, in their lexicographic definition order. So, for example, the C function below, called **snark**, declares three **auto** integers called **source**, **dest**, and **len**. These variables would be placed in workspace slots 0, 1, and 2 respectively (workspace slot 0 has the lowest memory address in the falling stack).

- Following the local **auto** variables, is the return address and the static link pointer. The static link pointer is used by the transputer's non-local load, store, and pointer instructions. With reference to **snark**, this would put the return address in workspace slot 3, and the static link in workspace slot 4. However, if the module used any static data, another slot is used as a static pointer to the other module.
- Finally, in ascending slot positions, comes the function parameter list, again in order of their lexicographic left-to-right declaration. So, for **snark**, parameter **a** occupies slot 5, **b** occupies slot 6, and slots 7, 8, and 9 go to parameters **i**, **j**, and **n**.

If the function has no local variable declarations, then the first parameter occupies workspace slot 2. This is why the **loc (a)** example above used the assembler command **ldl 2** to access the first parameter.

```
int snark (a, i, b, j, n)
char *a, *b;
int *i, *j, *n;
{
    int source, dest, len;

    source = b + (*j) - 1;
    dest   = a + (*i) - 1;
    len    = *n;

    asm {
        ldl 0 ; /* source */
        ldl 1 ; /* dest   */
        ldl 2 ; /* len    */
        move;
    }
}
```

A function like **snark** is used in [8], again called from a FORTRAN environment. The reason for the -1 offset in the initialization of **source** and **dest** is to do with the subscripting incompatibilities between C and FORTRAN *languages* (as opposed to an obscure feature of the INMOS scientific-language systems). This problem is further compounded in higher dimensions (as Dr Who frequently observes) due to the array column / row major allocation differences.

Review of how the transputer implements procedure calls

It is instructive at this point to consider how the transputer implements a function call / return. The **snark** function will be used as an example to show how the parameters are set up and how the workspace is used. Figure 11.5 illustrates the situation.

The transputer implements function / procedure calling with the **call** and **ret** instructions. The workspace pointer is adjusted using the **ajw** instruction. [5].

Consider the mechanics of a function call :

- The function that calls **snark** places all but two of the **snark** parameters at the bottom of its own workspace. In descending memory order, these are shown as **n**, **j**, and **b**. It then puts the other two parameters and the static link into the transputer's registers. Register C gets **i**, register B gets **a**, and register A gets the static link.
- The transputer's **call** instruction adjusts the workspace pointer, allocating four new positions into which it stores the three registers and the instruction pointer. This has the effect of placing the function return address, the static link, and all the function parameters contiguously in memory, as shown in Figure 11.5. The diagram shows the initial value of the workspace pointer, immediately following the call to the **snark** function — the return address (old instruction pointer) is at slot 0.
- The first action done by **snark** is to allocate workspace for its own **auto** variables. Since there are three, it does this with an **ajw -3**, which leaves the **snark** workspace numbered as shown in the

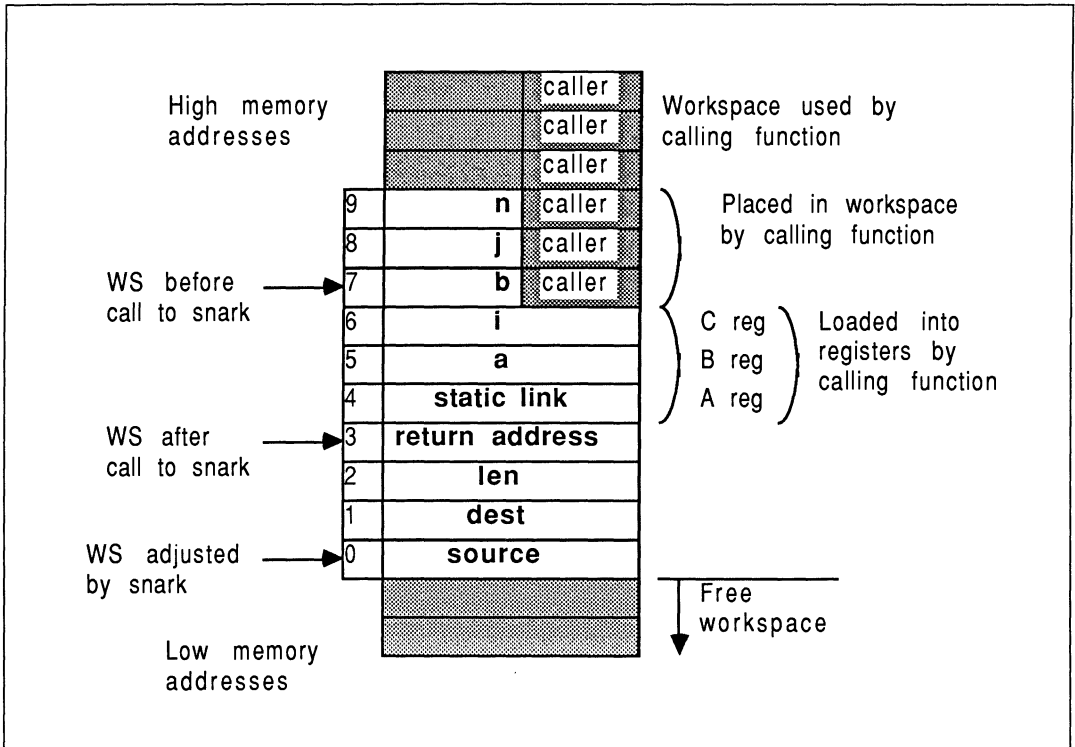


Figure 11.5 Function calls and workspace usage

Figure. The total stack workspace of **snark** is then ten words, of which the top three overlap with the workspace of the calling function. All the parameters are stored contiguously above the static link pointer, and all the local variables are stored contiguously below the return address.

- The last action of **snark** is to restore workspace used by the local function variables. This is done by an **ajw 3** instruction. This leaves the return address at slot 0 again. It is important to ensure that the workspace pointer has the value it had originally, immediately following the **call** instruction to the **snark** function.
- The **ret** instruction restores the instruction pointer to the value it had before the call to **snark**, and deallocates four workspace locations. This returns the workspace pointer to the value it had immediately preceding the **call**. Since **ret** does not corrupt the evaluation stack, up to three values can be returned to the calling environment.

The C assembler restrictions and capabilities

The V1.3 C compiler should not be used to symbolically access local variables or parameters — use the explanations given here as to where items will be placed in local workspace, and access them explicitly by slot number as in **snark**. Remember, the assembler insert feature in V1.3 C is not documented and not supported, so don't expect too much from it. However, both C assemblers will handle automatically any **pfix** and **nfix** instructions required to encode large values.

The Parallel C assembler allows symbolic access to parameters and local **auto** variables. **extern** variables can also be symbolically accessed but only within the scope that reserves storage for them. Individual statements within an **asm** directive cannot be labelled. Reference [6] should be consulted for the implementation capabilities of Parallel C.

11.3.12 Mixing occam and non-occam compilation units within the same process

There are many advantages to having a non-occam compilation unit call an occam PROC, rather than call another scientific-language procedure compilation unit. Firstly, the occam PROC requires no elaborate support from a run-time library. Secondly, occam PROCs are re-entrant because they have no concept of "writable static data", which means that occam PROCs and any of the occam library support procedures can be shared by any number of scientific-language processes on the same transputer. Thirdly, the occam support package is more mature and robust than any of the current INMOS scientific-language development systems.

In addition to the above discussions of the scientific-language compilation systems, some additional considerations are appropriate when involving occam PROCs. These include :

- Parameter type compatibilities between occam and non-occam systems.
- Hidden parameters required by occam PROCs.
- Array parameters.
- occam vectorspace support by non-occam compilation units.
- Calling occam FUNCTIONs rather than occam PROCs

These additional considerations are now explored :

Parameter type compatibilities

A working knowledge of the data storage and parameter passing mechanisms discussed above in the context of mixed-language scientific-language systems is useful when calling occam PROCs.

occam's **VAL** parameters correspond to C's non-pointer parameters, and Pascal's non-**VAR** parameters. In addition, occam **VAL** parameters which do *not* fit into a single machine word are expected to be passed by pointer reference. So, FORTRAN **DOUBLE PRECISION** real parameters would correspond to either a **VAL REAL64** or simply a **REAL64** parameter in occam. (Generally though, FORTRAN parameters are not in correspondance with occam **VAL** parameters).

C's pointer parameters, Pascal's **VAR** parameters, *any* FORTRAN parameters, and those parameters which cannot fit into a single machine word correspond to occam's non-**VAL** parameters.

Hidden parameters

Each scientific-language compilation unit passes, as a hidden parameter, the so-called static link pointer. This is a pointer to the static data for that compilation module. In occam this static link has to be accommodated by explicitly including a dummy integer first parameter in the formal specification of the occam procedure :

```
PROC occamproc (INT dummy, REAL32 other.parm)
```

This PROC can be called from C, Pascal, or FORTRAN, but the caller must not explicitly use two parameters in the calling specification.

Array parameters

C and occam enjoy totally compatible array allocation strategies, in terms of the storage mapping function, and array index subscripting. This is definitely not true of FORTRAN, which stores array dimensions in exactly the reverse strategy to occam, with wild and wacky possibilities as far as subscripting is concerned. It is not encouraged to access multi-dimensional arrays between either occam or C, and FORTRAN. [8] shows an example of the complications involved in accessing elements in a single dimensioned FORTRAN character array, from a C function.

In occam any unsized array strides in the formal specification of the PROC are in fact included as hidden

parameters, immediately following the pointer to the array parameter, in lexicographic left-to-right order of the missing strides. This means that a scientific-language compilation unit calling an OCCAM PROC with an unsized array must *explicitly* include parameters to specify the each unsized dimension. For example, the following OCCAM PROC specification

```
PROC occamproc (INT dummy, [BYTE other.parm)
  -- dummy holds the static link
  -- this PROC has hidden parm for size of other.parm
  -- call it explicitly with an extra INT parameter
```

must be called from, say C, like this :

```
char string[MAXSTRING];
... initialize the string
  occamproc (string, MAXSTRING);
```

Here, it is faster and safer to pass a pointer to the *whole* memory block reserved for the string, rather than do a run-time `strlen` for example.

Vectorspace

If the OCCAM PROC to be called has been compiled with vector space on, then it is necessary to explicitly pass to the PROC, as the last parameter, a word vector of a size sufficient to contain the vectors used by the OCCAM PROC. The pointer required should point to the base address of a sufficiently large contiguous memory area. This figure can be determined by using the D705B `ilist` utility on the compiled and linked OCCAM `.c%` file, with the `/e` entrypoint option; or alternatively from the compilation descriptor. Worked examples are included elsewhere in this document.

As an example, if the previous example was compiled with separate vector space on, and required 42 words of vector space storage, then the C must pass an extra final parameter :

```
char string[MAXSTRING];
int vectorspace[42];
... initialize the string
  occamproc (string, MAXSTRING, vectorspace);
```

OCCAM parameter supersets

In OCCAM timers, channels, and ports can never be **VAL** parameters. A timer parameter occupies no storage and so no parameter slot is reserved for it (this is also true for arrays of timers).

A **CHAN** type is represented by a pointer to the word containing the channel contents, which could be either a hard or soft channel.

Ports are represented the same way as the datatype for which they are a port. When a port is passed as a parameter, it is represented as a pointer to the corresponding data item.

Calling an OCCAM FUNCTION

All the discussions of OCCAM PROC parameter arguments apply to OCCAM FUNCTIONS, but with some additional complications. The recommendation to be given is to *never* directly call an OCCAM FUNCTION from a non-OCCAM compilation unit. Instead, call the OCCAM FUNCTION from a stub OCCAM PROC. Here's why :

For OCCAM FUNCTIONS returning a single result that can be accommodated in a single machine word, the result is returned in the transputer's A register (on a T414 or T425), or in the floating point A register on a T800 if the result is floating point. The first case here is compatible with where the C compiler expects to find function results.

However, for OCCAM FUNCTIONS returning more than one result, or where the single result does not fit in a single machine word, there is the additional complication of where to store the multiple results. This is in fact

achieved by passing hidden parameters to the FUNCTION arguments, which represent pointers to areas of memory where the results can be stored. The first three results that can be accommodated in a single machine word are returned in the transputer's A, B, and C registers. Other results require one hidden parameter per result, and on the T800, the floating point registers are not used at all to return values if there is more than one result. It's life, Jim, but not as we know it!

These hidden parameters for FUNCTION result storage *must* be placed at the very start of the explicit parameter list. The problem with calling non-OCCam FUNCTIONS directly from non-OCCam compilation units is that the static link is unavoidably passed in as the first parameter to the FUNCTION. This is no good because the FUNCTION could try to use it as a results storage area.

So, if one wishes to make use of OCCam FUNCTIONS from a non-OCCam compilation unit, and since you can't change the laws of physics, the recommendation is to call the FUNCTION indirectly from an OCCam PROC, and use non-VAL parameters to return the results to the calling environment, thereby circumventing all the difficulties described above. You *know* it makes sense ...

11.4 The INMOS D705B occam-2 toolset

The D705B OCCam toolset consists of an OCCam-2 cross compiler, an OCCam-2 syntax checker, a librarian, a linker, a binary lister, a bootstrap utility, a configurer, a makefile generator, a symbolic network debugger, a simulator, and the *iserver* file server / loader. In addition, some support for converting TDS software into toolset format is provided.

Code produced by the D705B is compatible at source and binary levels across the PC, VAX, and Sun-3 toolset platforms. All tools display usage information if invoked with no parameters, all tools have the same "work in progress" information selector (*/i*), and most can be re-run without reloading them. The file name conventions facilitate the use of automated tools to control the system generation of arbitrary transputer networks.

The remainder of this section discusses the D705B product OCCam-2 toolset. As each tool is discussed, the filename extensions employed at each stage will be shown in brackets. The % symbol is used as a single character wild-card in these filename extensions.

11.4.1 Software development using the D705B

Figure 11.6 shows a simple overview of the software development cycle using the D705B OCCam toolset software. Software implementation begins at the top of the diagram, and ends at the bottom. Rounded boxes represent specific operations, hexagonal boxes identify specific tools employed, and squared boxes represent real files such as libraries. The dashed line shows that the OCCam compiler accesses the (proprietary and user's) OCCam libraries at compile time, to check the procedure parameter interfaces across separately compiled units. The security afforded by this strict type-checking is part of the OCCam language specification, and is not offered by the scientific-language implementations.

In any software project, it is not possible to proceed down the diagram past any point until all the relevant operations shown above it have been done. Any operations shown horizontally adjacent can be performed at the same time. In broad terms, the software permits the OCCam and non-OCCam software for a transputer network to be developed concurrently by independent teams of programmers. At both source and binary level, the software developed will be compatible across PC, Sun-3, and VAX development platforms. A further advantage is that any development systems not available across the OCCam toolset development base, can still be used on their native machine and contribute binary object code for integration by the OCCam toolset on another platform. The D705B facilitates hooks for use with the programmers favourite version control and reconstruction software.

A typical application development scenario might look like this. Numbers refer to Figure 11.6. When all scientific-language source for a process is available, it is compiled and linked with run-time support. Once all such scientific-language object is available for a single transputer, and all OCCam source is available for that transputer, (point 1 in the Figure), the OCCam compiler is invoked. Immediately afterwards, at point 2, the toolset linker resolves external OCCam references by reading in the OCCam libraries specified, and merging all required code into a single object file that represents the process that runs on that transputer (point 2).

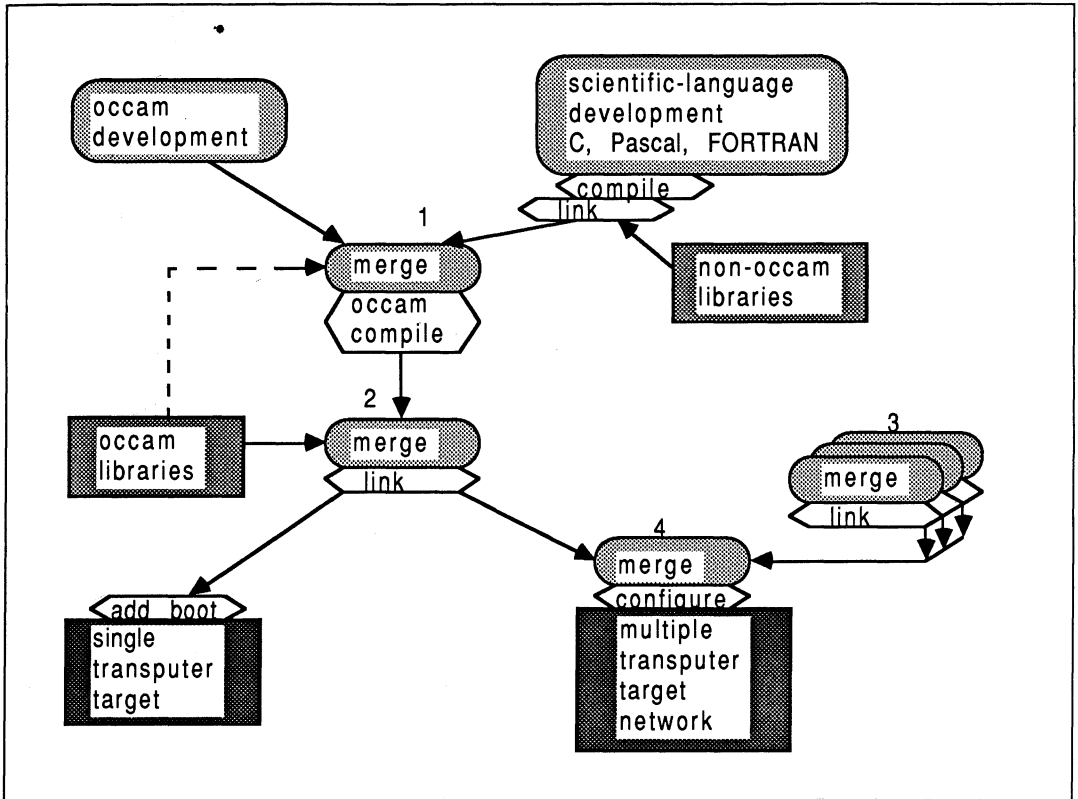


Figure 11.6 Overview of D705B software development

Only when this has been done for each unique transputer (point 3) can the system as a whole be realized (point 4).

In real-life, for a large project, one would place pre-compiled and pre-linked compilation units (derived from any language) into libraries that could be used by other parts of the system. One would also employ structured and methodical validation and verification techniques to components before bonding them together. The toolset's support for teams of programmers facilitates all stages of software implementation.

Because it is expected that teams of developers could be working on the same project, across potentially several development platforms, it is important to have a clear convention for identifying the contents of each file. This is achieved by using a homogeneous set of filename extensions. Because of the sophistication of the D705B, this requires a sizeable range of filename extensions, shown in the next section.

11.4.2 File naming convention

The file name extension convention for the D705B is extensive. For some files, the last two filename extension positions are dependent on the processor type and the error mode, explained in Sections 11.4.3 and 11.4.4.

File extension	Contents
<code>.occ</code>	occam source
<code>.inc</code>	include file of protocol or constant definitions
<code>.t%%</code>	separately compiled object code
<code>.l%%</code>	linker indirect command file
<code>.c%%</code>	linked code unit
<code>.s%%</code>	linker symbol table
<code>.m%%</code>	linker code map
<code>.b%%</code>	bootable code file for a single transputer
<code>.d%%</code>	descriptor file for a single transputer
<code>.r%%</code>	single transputer code with no bootstrap
<code>.lib</code>	library file
<code>.libb</code>	librarian build command file
<code>.liu</code>	library useage file (describes library nestings)
<code>.pgm</code>	OCCAM configuration description file
<code>.map</code>	configuration map
<code>.dsc</code>	configuration descriptor
<code>.dmp</code>	memory dump file
<code>.bt1</code>	link bootable file for transputer network
<code>.btr</code>	ROM bootable file for transputer network

Don't be put off by this horrific-looking table — its really seductively powerful once familiar. Simple calculation shows that there are over 200 different possible filename extensions, although not all of these are likely to materialize in a single project.

A word of advice : stick to these file name conventions, and be explicit with the filename extensions wherever possible. This will give you the maximum support from the automated system makefile generator (`imakef`).

11.4.3 Processor types

The compiler can produce code for the T212, T222, T414, T425, and T800 transputers. While all transputers are compatible at the OCCAM source level, some transputers are additionally guaranteed compatible at the binary T-code level. This compatibility is determined by the intersections of their instruction sets. To this end, the compiler can produce code that is guaranteed to run on a set of transputers :

Code set	Compatible processors
TA	T414, T425, and T800
TB	T414 and T425
TC	T425 and T800

The source restrictions on what can be compiled in each code set are determined by the instruction set intersection of the code class. Code set TA cannot contain any floating point, CRC, or 2D block-move. Code set TB can contain floating point (implemented in software by libraries), but not CRC or 2D block-move. Code set TC can support CRC and 2D block-move, but not floating point. Providing that the code produced for the different processors in a class would be the same for a given compilation unit, then that unit can be compiled in that class. All the 16-bit transputers (T212, T222, and M212) share the same instruction set, so the compiler makes no distinction.

These code sets are illustrated in Figure 11.7, which also shows the relationship between the processor classes and the basic processor types. The diagram shows that code compiled for processor types lower down in the tree can call code compiled for processor types above them *and* connected to them (possibly indirectly) by an ascending line. For example, T414 code can call T414, TB, or TA code, but TA code can only call other TA code.

To identify which processor (class) a given piece of code has been compiled for, the table above uses the % in the second position of the filename extension to indicate the processor type, which is one of **2**, **4**, **5**, **8**, **a**, **b**, and **c**.

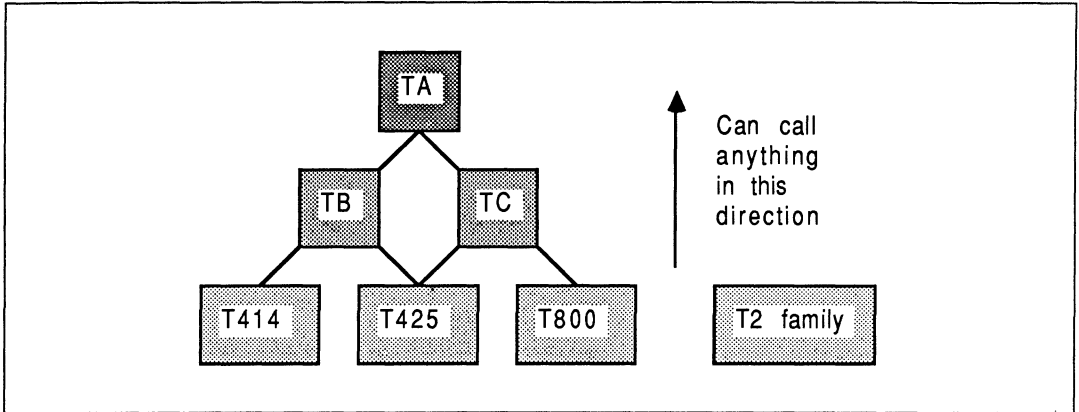


Figure 11.7 Processor compilation class hierarchy

If you compile code for any transputer *class* other than TB, the use of the compiler maths libraries must be disabled with the `/e` compiler option. This is because the compiler maths libraries are significantly different between the floating point T800 transputer, and the non-floating point transputers which are represented by class TB. So, classes TC and therefore TA encompass the floating-point and non-floating-point transputers, and therein lies the problem. The main differences arise because the T800 implements directly as instructions many functions which are represented as library calls for non-floating point transputers.

A further advantage of processor class compilation is that resultant libraries using generic code can be considerably smaller while still supporting a processor range. This technique will help to reduce the software size overheads of supporting present-day and future more powerful processor types.

11.4.4 Error modes

The compiler can produce code with differing behaviour when run-time errors occur. There are three error modes, suitable in different cases :

Error mode	Behaviour on error	Identity
HALT system	Total system halts	h
STOP process	Only errant process stops	s
UNDEFINED	Arbitrary effect	u

These are referred to as HALT, STOP, and UNDEFINED (REDUCED), and are identified with the letters **h**, **s**, and **u** in the last position of the filename extensions shown previously.

Each error mode is suitable in different situations.

- **HALT** : The default mode is HALT system mode, which is useful for developing and debugging a system. This mode is implemented using the transputers' `seterr` instruction following segments of code to be checked by causing an unconditional assertion of the error flag, or using in-line checks like `csub0`.
This mode is used in conjunction with a halt-on-error bootstrap, and run with the `iserver's` `/se` error test parameter.
- **STOP** : The STOP process mode ensures that errant processes do not communicate with other processes. This mode can be used to construct a system with software redundancy that exhibits "graceful degradation", allowing some operation even if parts of a system fail.

This mode is implemented using the **stoperr** instruction, which deschedules the current process if the error is set (but does not affect the status of the error flag). It is used in conjunction with the **testerr** instruction which loads *false* into the evaluation stack if the transputer's internal error is set, and *true* otherwise (it also clears the error flag). This mode produces the largest and slowest code, due to having to use **testerr** / **stoperr** pairs, rather than **seterr** instruction used in the previous execution mode.

- **UNDEFINED** : The UNDEFINED (REDUCED) error mode should only be used for optimising programs that are known to be correct, because the amount of run-time checking included by the compiler is minimal. In this mode, invalid processes have an arbitrary effect. Code compiled in this mode is the most compact and fastest, compared to the other two error modes.

There is an additional error mode called UNIVERSAL, identified by **x**. This is implemented in the same way as UNDEFINED, with minimal checking. Separately compiled units compiled in this mode can be called from units in any of the other error modes, and may call other units compiled in **x** mode. This is shown in Figure 11.8. The general rule is that all separately compiled units must be compiled in the same error mode. These error modes are described more fully in [2].

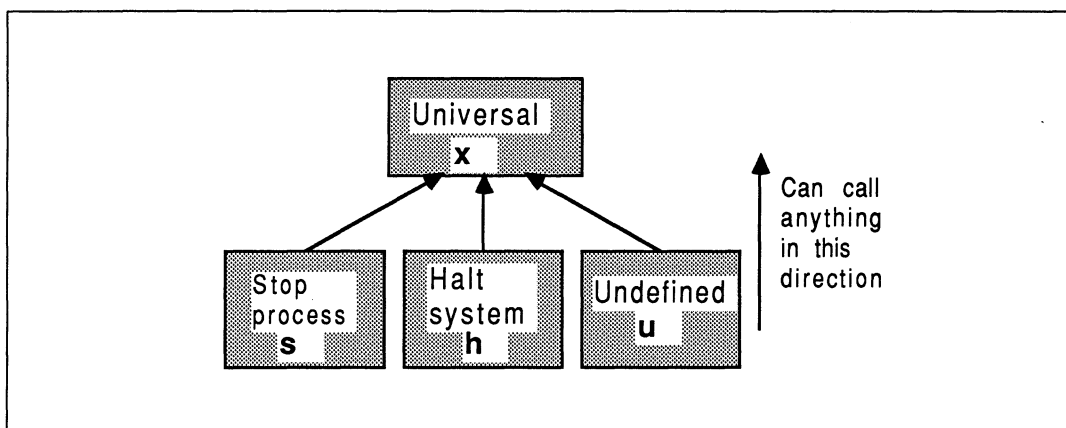


Figure 11.8 Processor error mode hierarchy

If code is to be compiled in UNIVERSAL error mode, use of the OCCAM compiler's libraries *must* be disabled with the **/e** option. This is because the compiler libraries exhibit different behaviour in different error modes, so it is not possible to use floating point, extended data type and other compiler library functions with the UNIVERSAL error mode.

11.4.5 The makefile generator

The **imakef** utility automatically generates a makefile to rebuild a multi-transputer program, a single transputer program, or a library. The C source is supplied so that users can adjust the program for similar tools. The program will also generate linker command files and library usage files. The program does not produce any rules for object code that has been imported using the **#IMPORT OCCAM** compiler directive, although it does assume that any linked code referred to is derivable ultimately from OCCAM source files.

11.4.6 The OCCAM compiler

The compiler **occam** is a full OCCAM-2 compiler, supporting **FUNCTIONS**. OCCAM source is placed in **.occ** files, and compiled object is stored in **.t%&** files.

The **#USE** directive is used to reference separately compiled units from within OCCAM source text. The **imakef** utility ensures that certain rules surrounding **#USE** are observed, in connection with non-circularity of references, compilation before usage, and compatible processor types and error modes. The default suffix

with **#USE** is `.t%` for compiled units, depending on compiler options, and `.lib` for libraries.

The **#SC** references a separately compiled unit, and is included only for compatibility with the INMOS TDS. It is recommended that the **#USE** directive is instead employed to reference separately compiled procedures, as this removes the constraint on specific ordering of separately compiled units at link time. (SCs must be linked in a special order because the OCCAM compiler generates direct calls to the SCs, rather than allowing the linker to patch them. To do this, the compiler must assume they are loaded in a specific way). Simple substitution of the directive **#USE** for the **#SC** directive is sufficient.

The **#IMPORT** directive takes the filename of the compiled and linked non-OCCAM application, to allow the **imakef** utility to handle non-OCCAM aspects of a system. This also serves to conceal unpleasant detail concerning the instantiation of non-OCCAM processes, while presenting to the OCCAM compiler something that *looks* like an OCCAM **PROC**.

An additional **#COMMENT** directive allows a comment string to be associated with the compilation unit, intended to hold the version number, date of last update, and a short description.

The directory path in which a referenced file resides can be specified explicitly, or relative to the directory in which the compiler was invoked, or have no path specified. It is strongly advised, especially in multi-platform toolset development, that no directory path specifications are ever included in OCCAM source directives. This would have the effect of compromising the source-level portability amongst platforms on the Sun-3, VAX, and PC. To circumvent this, a sequence of directory paths which will be searched can optionally be specified by using the PC environment variable **ISEARCH**. There are equivalent path specifications in the other toolsets, and these should represent the *only* host-specific parts of toolset development.

The default is to compile OCCAM for a T414 in HALT-system compilation mode, with separate vector space, alias and usage checking enabled. This gives a `.t4h` object file.

11.4.7 The syntax checker

The OCCAM compiler stops when it detects the first error. At times, it is more useful to have a list of errors available to permit bulk editing operations on virgin source. The syntax checker **icheck** generates such a list of errors, and has particularly good error recovery due to the fixed format of the OCCAM language.

11.4.8 The librarian

The librarian **ilibr** is used to collate separately compiled units into a single library file (`.lib`). Libraries can be built from units compiled for mixed processor types and error modes. They provide a convenient unit for distributing collections of procedures and functions in a single file. Libraries form the basis for the selective loading mechanisms of the linker (The linker will selectively load separately compiled units from a library only if they satisfy an outstanding reference and match the processor type and error mode requirements). Indirect files can be used to list the names of files to be included in the library.

A specification describing what object files have to go into a library is provided in a `.libb` file. One can specify compiled object and linked object files, for a range of processors and error modes. Note that it is not possible to mix source and object in the same file, so for example it is not possible to have OCCAM source **INCLUDE** files in a library.

The librarian also supports building libraries from units compiled with the scientific-language compilers. OCCAM procedures and functions are re-entrant and can be shared, through libraries, by separate parallel threads of execution on a single processor. As not all modules in the scientific-language libraries are re-entrant, the libraries as a whole are not re-entrant. This requires that separate copies of the libraries are linked with each scientific-language process.

Libraries may reference other libraries, but may not reference code via a **#SC** directive. This is because the positioning of SC code is critical, whereas the library mechanisms locate code in arbitrary places.

The librarian ensures the integrity of the library by checking each new addition for violation of uniqueness of processor type and error mode within the library.

11.4.9 The linker

The linker **ilink** composes a collection of separately compiled units, (**.t%%** and **.bin** and **.c%%** linked units) resolving external references, to give a single code unit (**.c%%**). This is typically used to build the program code for a single processor. The output of the linker is in the form of a separately compiled unit, like that produced by the OCCAM compiler, which means that linker output can be re-submitted as input at a later linking stage.

The first argument in the link list is always a separately compiled unit, not a library. This defines the processor target type, error mode, and entry point for the linked unit, and all further units must be compatible with respect to this processor target (set) and error mode.

Separately compiled units in the argument list are loaded unconditionally, but units in libraries are loaded only if they match the processor type and error mode of the first argument, and if they satisfy some outstanding reference. The processor target rule specifies that units may call units with at least as general target set (so T800 units can call TA and TC units, for example). The error mode rule is that units may call units with at least as general error mode set (so HALT, STOP, UNDEFINED, and UNIVERSAL may call UNIVERSAL, but HALT may only be called from HALT).

If the **#SC** directive is used to reference separately compiled units, then these units must be linked in the correct order. The **imakef** utility will generate the linker command file to achieve this correctly.

There are some restrictions as to how the linker can be used with scientific-languages. Only complete scientific-language programs can be linked using the linker — this is because the linker has to resolve the initialization chain for the scientific language compilers. To do this, it has to associate an entry point name with the output file it produces, and this is only meaningful for a complete scientific-language process. Multiple scientific-language processes to run on a single processor may be individually prelinked with run-time support and resubmitted to the linker with the main OCCAM calling process.

Linker control input may be re-directed from a specified file or standard input. However, re-directed linker command input may *not* itself be re-directed. Therefore, an indirect file may not refer to another indirect file or to standard input. Several indirect files can be specified on the linker command line. Command options can be placed in the linker indirect file, for example, to optimize the positions of certain symbols.

11.4.10 Binary lister

The binary object lister **ilist** is used to generate documentation information from binary files, either from separately compiled units or from library files. Various command-line options permit different types of documentation to be produced. The options are accumulative, so that more than one type of output can be requested with a single command. Information concerning modules, procedures within them, entry points, processor types and error modes, external references, and workspace requirements can be extracted from any binary object file (**.bin**, **.lib**, **.c%%**, **.t%%** etc).

11.4.11 The bootstrap tool

The **iboot** utility prepends bootstrap and loading code to a program for a single processor. The input file will have been produced by the linker (**.c%%**), and the output file can be executed on a transputer (**.b%%**) using the server (**iserver**). The default bootstrap will halt the processor if the transputer error flag becomes set. Optionally, the bootstrap will not halt the processor if the transputer error flag becomes set.

If the execution mode of the input object file is either HALT or STOP process, then the halt-on-error flag is set by the bootstrap code; otherwise the halt-on-error flag is not set in the bootstrap loader code. This, in conjunction with the type of bootstrap prepended, defines the program's behaviour if the error flag becomes set.

11.4.12 The configurer

The **iconf** configurer is used to create multi-transputer programs (**.bt1** or **.btX**), specified in a configuration description (**.pgm**), by using output from the linker (**.c%%** files). The configurer generates loading

and bootstrap information for a transputer network of arbitrary topology and composition. The bootstrap and loading information is complex due to the possibility of different transputer types in the network, each with potentially different amounts of memory.

The toolset configurer allows multiple processes to be **PLACED** at configuration level. In addition, any OCCAM that does not involve library references can be expressed at configuration level.

Network description information (`.dsc`) is also created for use by the debugger tool.

11.4.13 The debugger

The toolset debugger `idebug` allows a symbolic post mortem analysis of an arbitrary transputer network. Facilities exist to examine the contents of memory symbolically and in many different representations. The processes on the run-queues and timer-queues can be identified. It is possible to symbolically "walk down links" to processes operating at different ends of a channel (whether soft or hard). The debugger will locate to the source line at which the transputer error flag became set, allowing variable inspection. The procedure calling sequence can be traced back, also through libraries.

In the case of scientific-language debugging, the debugger can locate to the source line at which the transputer halted. This is possible in a mixed language system of arbitrary complexity. It is not possible to use symbolic debugging facilities in scientific-language source file because the scientific-language compilers do not produce sufficient information for the debugger. However, procedure trace-back is still possible within this framework.

Later sections in this document discuss how best to use the debugger with scientific-language systems.

11.4.14 The simulator

The toolset simulator `isim` can run almost any program that can be run on a single T414 transputer, on a boot-from-link evaluation board. The simulator provides most of the symbolic debugging facilities provided by the toolset debugger, plus the ability to set break and watch points at source level, and single-step a program. An important feature of the simulator is that the compiled code is exactly that which can be booted onto the transputer board and run normally.

Unfortunately, the simulator cannot accommodate non-OCCAM components. The simulator is not discussed further in this document.

11.4.15 Supplementary tools

There are a number of utility tools supplied with the TDS which are also supplied with the toolsets. In particular, the tools for EPROM and memory interface programming, and the transputer network tester, are provided.

11.5 Handling non-OCCAM processes

The previous sections have presented information concerning the INMOS scientific-language systems, and the D705B OCCAM toolset. Now, this information will be combined to show how to correctly integrate non-OCCAM processes within an OCCAM framework. The methodology of arbitrarily interconnecting non-OCCAM processes is known as *equivalent OCCAM process* technology (EOP).

11.5.1 Equivalent OCCAM process technology

The scientific-language systems create processes which can be made equivalent to an OCCAM process. The interface to these processes was devised for flexibility, and is not suitable for direct inclusion into a parallel system. The language-independent interface affords a general bilateral communication between a scientific-language process and an OCCAM process, while accommodating a certain flexibility in the workspace arrangements. It should always be wrapped in a layer of OCCAM which exposes only conventional OCCAM channel parameters to the outside world.

There are three basic forms of equivalent OCCAM process (EOP) which can be built :

- Type 1 : Used when a program runs on a single transputer communicating only with the host server.
- Type 2 : Used when the program communicates with other processes as well as the host server.
- Type 3 : Used when the program communicates with other processes but does not communicate with the host server.

To form an EOP from a C, Pascal, or FORTRAN program, the object modules comprizing the program (including the run-time library) are linked with special OCCAM interface code, using the toolset linker **ilink**. These interfaces conceal various supporting details, and offer a fixed language-independent interface to OCCAM. INMOS supplies interface code for the three types of EOP described above.

The Type 1 interface

A Type 1 interface is used for programs communicating only with the host server **iserver**. This is equivalent to the standard OCCAM harness used by the scientific-language development systems. The Type 1 interface has the following parameters :

```
PROC MAIN.ENTRY (CHAN OF SP fs, ts,
                 []INT free.memory,
                 []INT stack.memory)
```

The channels **fs** and **ts** communicate from and to the host server **iserver**, using the protocol **SP** defined in a standard library (not shown). The **free.memory** vector is used as program workspace. If the size of the **stack.memory** vector is zero, then **free.memory** is used for the run-time stack, heap, and static workspace. Otherwise, the **free.memory** is used for heap and static workspace. The DOS environment variable **IBOARDSIZE** specifies the size of **free.memory**; its read at run-time by the bootstrap loader. The **stack.memory** is used as run-time stack storage if the size of the vector is not zero. Its size is determined when the bootstrap is prepended by the **iboot** tool, using the **/s** option.

The code for **MAIN.ENTRY** is contained in the files **maintent.c%%**, depending on the transputer type and error mode required. The programmer does not have to write any OCCAM for this interface.

To use this interface, consider the following example to build a T414 program in UNDEFINED error mode. A list of compiled program object binaries (including run-time libraries) is placed in the linker control file **proglink.14u**. The required linked output is to be placed in file **cprog1.c4u**, then bootstrapped with a 512 word run-time stack vector. The D705B operations required are :

```
ilink maintent.c4u /f proglink.14u /o cprog1.c4u
iboot cprog1.c4u /s 512
```

The Type 2 interface

A Type 2 interface is used for programs communicating with other processes as well as the host server. This interface is used with non-OCCAM programs linked with the full versions of their run-time libraries. The Type 2 interface has the following parameters :

```
PROC PROC.ENTRY (CHAN OF SP fs, ts,
                 VAL INT flag,
                 []INT ws1, ws2,
                 []INT in, out)
```

The channels **fs** and **ts** communicate from and to the host server **iserver**. The **flag** is used in conjunction with the workspace vectors **ws1** and **ws2**. If **flag** is zero then **ws1** is used as the run-time stack and **ws2** is used for statics and the heap. If **flag** is 1 then **ws1** is used as a combined stack/heap/static workspace. Vectors **in** and **out** are used as pointers to OCCAM channels going to and coming from the non-OCCAM process.

The code for **PROC.ENTRY** is contained in the files **procent.c%%**, depending on the transputer type and

error mode required. To use this interface, a simple OCCAM harness of the type below is written to bind the channels used by the server and the other processes to a clean procedural interface :

```
PROC p.EOP2 (CHAN OF SP fs, ts,
            CHAN OF ANY from.outside, to.outside)

#IMPORT "cprog2.c4u"
[3]INT in , out:
[1024]INT stack.vector :
[5000]INT heap.vector :
SEQ
  -- establish user input and output channels
  LOAD.INPUT.CHANNEL (in [2], from.outside)
  LOAD.OUTPUT.CHANNEL(out [2], to.outside)

  -- EOP2 is the entry point name in cprog2.c4u
  EOP2(fs, ts, 0, stack.vector, heap.vector, in, out)
:
```

The **#IMPORT** command references the file name containing the linked EOP object binary file, its run-time library, and the Type 2 interface code. The channel pointers are initialized using the predefines **LOAD.INPUT.CHANNEL** and **LOAD.OUTPUT.CHANNEL**. 1024 words have been allocated for the stack, and 5000 words for the heap/static area. EOP workspace is required by the scientific-language process and the run-time libraries, and must be large enough for all of the run time stack, static data, and the heap used by the program and its libraries. As a rough guide, a minimum of 4000 words for static & heap workspace, and a minimum of 400 words for the run time stack, is advised. By the time an EOP is ready to commence, having been through the initialization sequence controlled by the run-time library, almost 100 words of stack space have already been used.

It is important to emphasize that this OCCAM harness is completely *standard* for a Type 2 interface. In the last line in the example above, the **EOP2** is the substituted name for the **PROC.ENTRY** defined. The name-change occurs at link-time, allowing any number of EOPs in a system to use the same interface code :

```
ilink EOP2=procent.c4u /f proglink.l4u /o cprog2.c4u
```

This has the effect of creating a linked file called **cprog2.c4u** which is **#IMPORTed** into the OCCAM harness above. From there onwards, the procedure **p.EOP2** is considered as a standard OCCAM procedure in the system — but it must always connect to the server.

The Type 3 interface

A Type 3 interface is used for processes that do not need to communicate with the host server. There are three types for use with C, Pascal, or FORTRAN programs linked with the reduced version of their run-time libraries :

- C programs

```
PROC PROC.ENTRY.RC (VAL INT flag,
                   []INT ws1, ws2,
                   []INT in, out)
```

- Pascal programs

```
PROC PROC.ENTRY.RP (VAL INT flag,
                   []INT ws1, ws2,
                   []INT in, out)
```

- FORTRAN programs

```
PROC PROC.ENTRY.RF (VAL INT flag,
                   []INT ws1, ws2,
                   []INT in, out)
```


Another Type 3 interface is used with C, Pascal, or FORTRAN programs that have been linked with the *full* version of the run-time libraries. This is called the *stub* interface. Normally, EOPs linked with their full run-time library would require a connection to the host server, preventing their use in a “remote” position. But the stub interface obviates this.

```
PROC PROC.ENTRY.STUB (VAL INT flag,
                     []INT ws1, ws2,
                     []INT in, out)
```

These interfaces take parameters with the same meaning as the Type 2 interface. Depending on processor and error mode, the C interfaces are stored in files `procentc.t%%`, the Pascal interfaces are stored in files `procentp.t%%`, and the FORTRAN interfaces are stored in files `procentf.t%%`. The stub interfaces are in `procents.c%%`. They are used in exactly the same way as the Type 2 interfaces. A simple template harness is written (exactly the same as for the Type 2 interface, but without the server channels), and the linker is used to change the entry-point name. For example, a Pascal program for a T800 in HALT error mode, to be instantiated with the identifier `EOP3` would be linked as follows :

```
ilink EOP3=procentp.t8h /f proglink.l8h /o pprog3.c8h
```

The corresponding `#IMPORT` for this would refer to file `pprog3.c8h`. An example of a Type 3 EOP is given in Section 11.7.2.

The most common arrangement in a multi-process system is for one Type 2 interface (communicating with the server), and the remainder are all Type 3.

11.5.2 D705B Processor classes

Concerning scientific-language processes, the EOPs cannot be compiled for a general processor class (i.e. TA, TB, TC), and therefore cannot be called by code compiled for a general processor class. This has an implication for library usage. For example, TA OCCAM harness code cannot call T414 EOP code. TA code can *only* call TA code. So, if one wishes to place OCCAM harness parts into a library as well as the linked EOPs, they must be compiled for either T414 or T800 execution.

11.5.3 EOP Startup and shutdown overheads

Each time an EOP is instantiated, there is a timing penalty to be paid. The nature and magnitude of this penalty depends on whether the non-OCCAM process is using the host file server facilities provided by the full run-time library, or whether the EOP is using the standalone run-time library for the language concerned. In either case, the EOP instantiation overheads are *enormous* compared to calling an OCCAM procedure. An understanding of these penalties is useful in deciding how finely to partition a non-OCCAM system into individual parallel processes. Both these cases are discussed below :

- EOP using the full run-time library

On a 20 MHz transputer, the time taken for an EOP to startup to be in a state capable of doing useful work varies from 25 to 40 *milli*-seconds, depending on the language. The start-up overheads in this case are partly concerned with run-time initialization of static data for each module in the EOP. Also, the start-up routines attempt to open the standard input, output, and error channels to the keyboard and screen. This involves dealing with the host file server, and accounts for the bulk of the time spent for most reasonably sized EOPs. This is clearly not the sort of thing to do too often — once an EOP is running, don't terminate it with a view to restarting it regularly!

There is also a timing penalty in shutting down an EOP. This is usually of lesser consequence than the startup overhead. In the shutdown period, any open files and streams are closed, which again involves dealing with the host file server. This is again typically 25 to 40 milliseconds, although it can be less than 10 milliseconds in unusually trivial cases.

- **EOP using the standalone run-time library**

For an EOP using the standalone run-time library, none of the penalty associated with communicating with the host file server is incurred. This typically results in start-up and shut-down penalties an order of magnitude smaller than those using the full run-time library. In other words, expect to spend between 1 to 4 milliseconds in starting and stopping each EOP in this way.

A corollary of this is that EOPs should only be used to perform fairly sizable units of work, compared to the overheads in instantiating and terminating them. It is important to be quite clear that once instantiated, the operation of the normal function / procedure / subroutine calls in EOPs is every bit as efficient as for compiled OCCAM. Calling an embedded heterogeneous compilation unit from within another compilation unit incurs no additional temporal penalties.

11.5.4 Practical considerations for writing harnesses

In writing custom harnesses, either as EOPs or as the top-level OCCAM on a transputer, there are several factors one can control. For example, the size and placing of stack and heap workspaces, board size considerations, and run-time specifications can all be used to advantage.

These issues are discussed below, after reviewing how the single-processor standard OCCAM harness supplied with the scientific-language systems is implemented.

Memory allocation by the standard scientific-language harness

In the INMOS scientific-language systems, all memory allocation is under control of OCCAM procedures. The INMOS scientific-language compilers employ a common model of memory usage. This enables the outputs from all compilers to be linked and loaded with the same tools, and also facilitates some mixed-language operations.

Using the Type 1 interface for an EOP on a single processor, the workspace allocated from the **free.memory** vector extends from the top of the OCCAM vector space zone to the top of the board memory. This memory area is shown in Figure 11.2 as unallocated memory. The size (in bytes) of the board in use is specified by the DOS environment variable **IBOARDSIZE**. Figure 11.9 shows how the unallocated memory is used by the Type 1 interface.

From **Mint** onwards, the OCCAM compiler that compiled the "standard harness" to support a single EOP, can allocate workspace. Using techniques described in [9], the compiler places a block of 512 words as low down in memory as possible. This memory block is reserved for a run-time stack for an EOP, and is mostly on-chip. Figure 11.9 shows this reserved run-time stack area in the OCCAM scalar workspace zone. On a T414 transputer, this uses up all the on-chip RAM. Even if the user does not run the application to make use of this stack, this memory is always reserved when using the standard OCCAM harness². There will also be a few words of scalar workspace required by the OCCAM process which instances the EOP.

With a single combined vector for workspace, the **free.memory** vector establishes the amount of memory available. As the size of this is determined at run-time using a DOS environment variable, the application always has access to the most workspace available. This obviates the need to re-compile an application to take full advantage of a larger / smaller board. If **IBOARDSIZE** is set too large, the run-time stack would be placed off the end of the board; if **IBOARDSIZE** is set too small then not all of the board's memory is availed.

Directly following the OCCAM scalar workspace (and EOP stack reserve) is the code for all the component modules in the non-OCCAM application and the OCCAM calling process. This includes OCCAM and non-OCCAM library modules. The linker will decide in what order each component part should be linked. By referencing any compiled OCCAM in an application referenced with **#USE**, the linker is free to select an arbitrary loading map for each transputer.

Immediately above the code is the non-OCCAM initialized static data area.

²The Parallel C and Parallel FORTRAN development systems do not reserve a block of 512 words for stack space unless instructed to do so. This means that even on a T414, the standard harness has an opportunity to place some code on-chip.

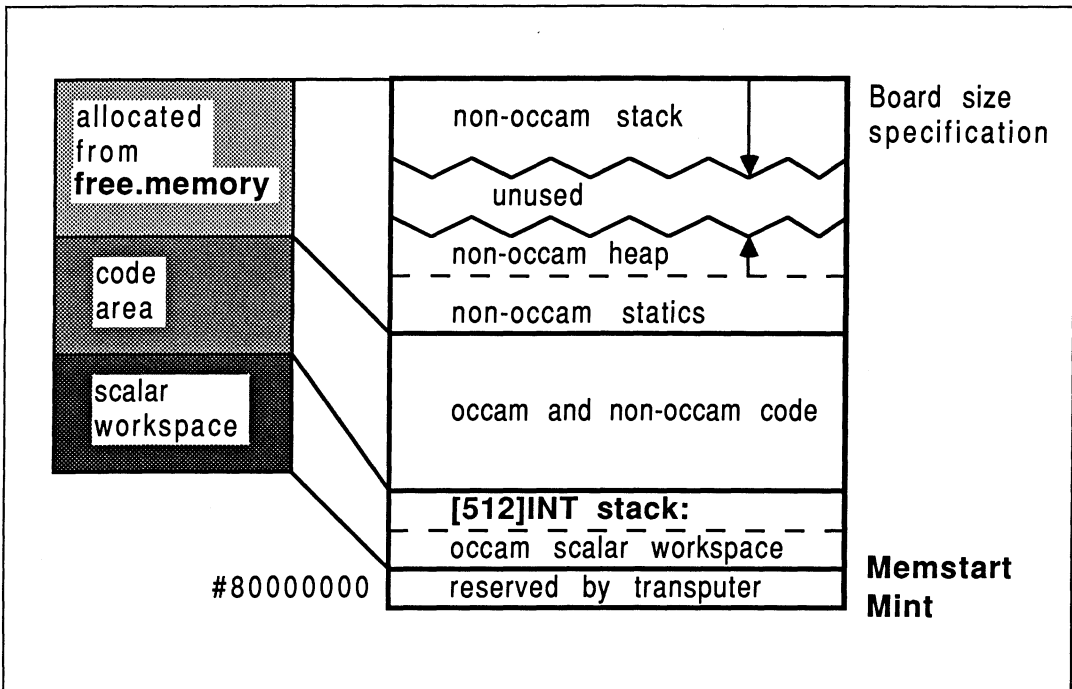


Figure 11.9 The scientific-language compiler memory map

Writing harnesses to allocate scientific-language workspace memory

When writing a harness, one can allocate workspace for the scientific-language systems from OCCAM vectorspace, rather than from the `free.memory` parameter. This would be the preference in two cases; first when one is writing a compact EOP harness, and second when one is writing harnesses for a transputer network (`free.memory` is not available in multiple processor systems).

One scientific-language process

The memory allocation for the system shown in Figure 11.9, has been instead allocated from OCCAM vector space, as shown in Figure 11.10.

This figure shows that, providing the OCCAM harness is compiled with separate vector space on, then the stack and heap areas sit lower down in memory than before (but still *above* the code zone). Suitable D705B OCCAM to implement a Type 3 interface like this is :

```
[50000]INT heap.vector :
[512]INT stack.vector :
PLACE stack.vector IN WORKSPACE :
program (0, stack.vector, heap.vector, in.EOP, out.EOP)
```

To increase the chances of placing the stack-vector (mostly) on-chip, the OCCAM harness to implement this would have to be compiled with vector space off (in which case the main static / heap workspace would sit *below* all code, or with vector space on the stack vector would be explicitly **PLACED IN WORKSPACE**. This latter case corresponds to Figure 11.10 and the OCCAM fragment above.

Notice that if the application will *definitely* not require the use of a separate run-time stack, one need not reserve any memory for it in a custom-harness. This will save on overall memory requirements, and allow the code to be placed lower down in memory.

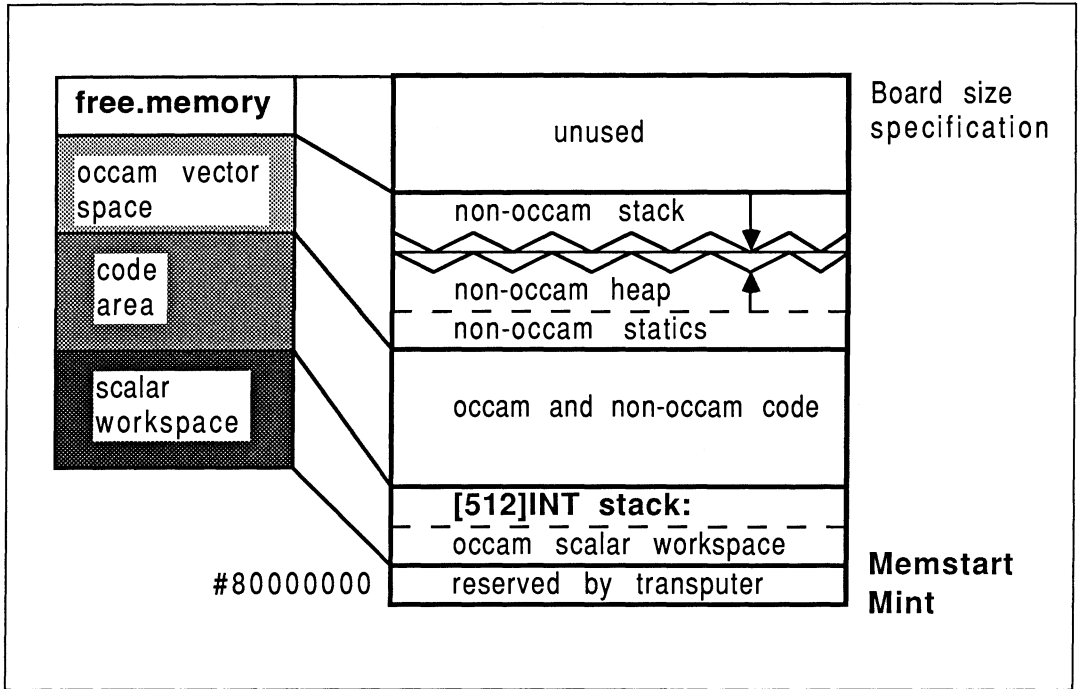


Figure 11.10 Allocating memory from OCCaM vector space

In a single transputer system, the **free.memory** parameter is still available; but it is unused and will be smaller than before since there is a much larger OCCaM vector space content. In a multiple transputer system, the **free.memory** parameter is *not* available, so harness techniques like those discussed here must be understood and employed by the performance-conscious programmer.

Two scientific-language processes

In a more general case, applicable to a single transputer and to an arbitrary transputer in a network, consider placing two scientific-language processes on a transputer. Following the guidelines above, one must allocate workspace for the EOPs by using OCCaM vectors (remember that the **free.memory** vector is not available in a network). One would normally compile the OCCaM harness with vector space on, thereby placing the workspaces above all loaded code, but remembering to explicitly **PLACE** the stack vectors **IN WORKSPACE**. In Figure 11.11, this case is illustrated.

D705B OCCaM to implement this memory arrangement (as a pair of Type 3 interfaces) is shown below :

```

PAR
  [50000]INT heap.vector2 :
  [512]INT stack.vector2 :
  PLACE stack.vector2 IN WORKSPACE :
  EOP2 (0, stack.vector2, heap.vector2, in.EOP2, out.EOP2)

  [50000]INT heap.vector1 :
  [400]INT stack.vector1 :
  PLACE stack.vector1 IN WORKSPACE :
  EOP1 (0, stack.vector1, heap.vector1, in.EOP1, out.EOP1)

```

Because the OCCaM compiler places the most recently declared variables in the lowest memory locations, this OCCaM and Figure 11.11 shows that the EOP1 stack is placed closer to **Memstart** because it is declared

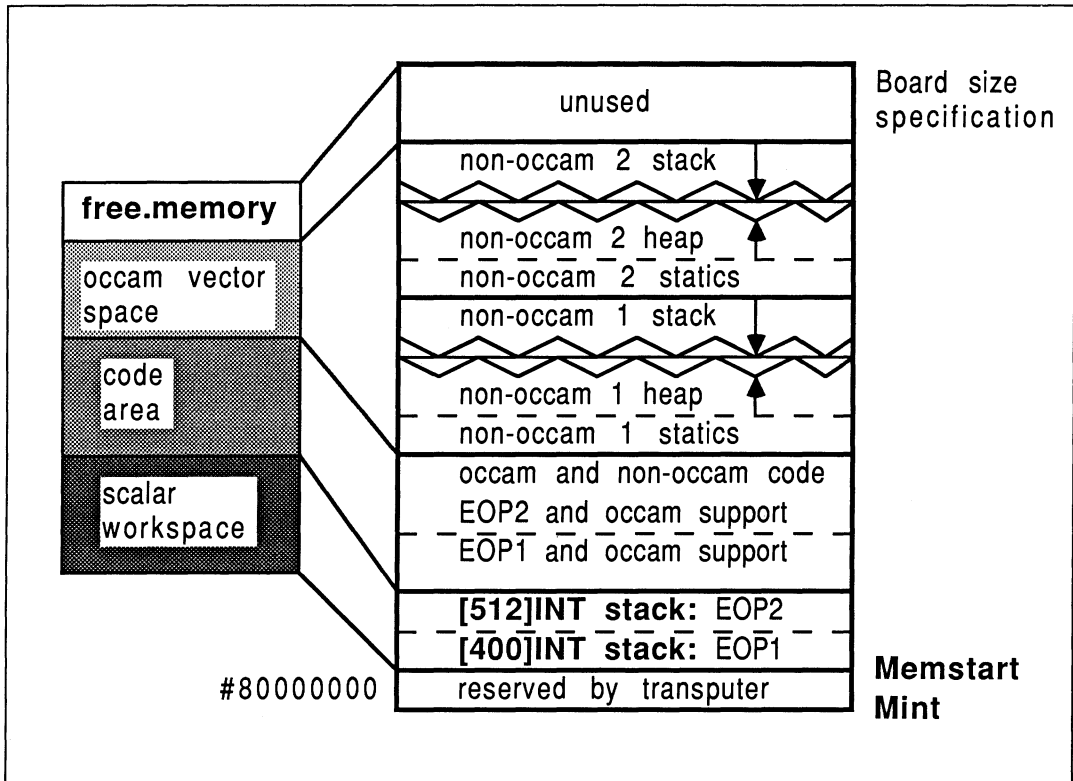


Figure 11.11 Allocating memory for two EOPs from OCCAM vector space

after EOP2. The stack for EOP1 is also smaller than that of EOP2, which would have been empirically determined as per Section 11.5.4.

Placing all EOP stacks below the code

It is usually worth compiling the OCCAM harness with vector space on, and explicitly forcing stack vectors to be placed in **WORKSPACE**. This has the effect that all EOP stacks are placed below the code area. Although it is unlikely that *all* such stacks could be accommodated on-chip, some board products such as the INMOS B404 module have a region of faster static memory below a large but slower dynamic store, and this software technique would allow the most suitable use to be made of this fast memory block without adjusting the software or re-compiling it.

Establishing EOP workspace requirements

INMOS do not provide any tools to allow one to estimate the size of stack or heap workspace required by an EOP. There is no simple way to determine the requirements for workspace, but the following comments might be useful in fine-tuning workspace sizes :

- When developing and testing an EOP, use one large combined stack and heap workspace. This is because there is less chance of an EOP running out of workspace if one allocates a total amount for stack and heap, compared to explicitly defining the sizes of these independently.
- As a rough guide, allow a minimum of 400 words for a separate stack area, and a minimum of 4000 words for static / heap area, for each EOP. Even during EOP start-up, at least 80 words from

the stack are used. The D705B toolset `ilist` object lister can be used to indicate the amount of initialized static workspace required by the linked EOP — this could guide one's heap workspace sizing estimates.

- The *actual* amount of memory used in any workspace by any given execution can be established by adding some extra pieces to the occam harness. By initializing all the elements in the stack and heap workspaces of an EOP to some value before instancing, as the EOP executes, the pattern will be over-written. This allows the extent of each workspace to be established, and can be done for each EOP in the system one by one. Remember that heaps grow upwards and stacks fall downwards.

Suitable OCCAM PROCs to perform the size estimation on stack and heap workspace areas are shown below :

```

-- stack and heap workspaces to be fine tuned
[512]INT stack.ws :
[50000]INT heap.ws :

PROC init.vec ([]INT vector, VAL INT pattern)
  SEQ i = 0 FOR SIZE vector
    vector[i] := pattern
  :

PROC used.in.stack ([]INT stack.ws, VAL INT pattern,
                   INT used)
  -- stacks fall down so scan upwards from element 0
  BOOL found :
  INT loop :
  SEQ
    found := FALSE
    loop := 0
    WHILE (NOT found) AND (loop < (SIZE stack.ws))
      IF
        stack.ws[loop] = pattern
        loop := loop + 1
      TRUE
    found := TRUE
    used := (SIZE stack.ws) - loop
  :

PROC used.in.heap ([]INT heap.ws, VAL INT pattern,
                  INT used)
  -- heap grows upards so scan from top element downwards
  BOOL found :
  INT loop :
  SEQ
    found := FALSE
    loop := (SIZE heap.ws) - 1
    WHILE (NOT found) AND (loop >=0)
      IF
        heap.ws[loop] = pattern
        loop := loop - 1
      TRUE
    found := TRUE
    used := loop
  :

```

One would then structure one's top-level harness like this :

```

PROC application(CHAN OF SP fs, ts)
  VAL INT pattern IS #55555555 :
  INT heap.used, stack.used :
  WHILE TRUE
    SEQ
      -- initialize workspaces
      init.vec(stack.ws, pattern)  -- preset stack vector
      init.vec(heap.ws, pattern)   -- preset heap vector

    PAR
      ...   Execute all application

      -- determine stack and heap usage
      used.in.stack (stack.ws, pattern, stack.used)
      used.in.heap (heap.ws, pattern, heap.used)

      ... report findings and terminate
  :

```

Obviously, to have significant meaning, this methodology would have to be repeated many times to thoroughly exercise the EOP. One would then leave a suitable (large) safety margin. Each EOP in a system would be tuned in this way, one at a time.

- If the D705B is involved, the same technique can be easily used for EOPs on any transputer because the debugger can be used to examine the workspace vectors after run-time. Use of the debugger in this technique only requires that all elements are pre-initialized to some identifiable value. Section 11.6 explains how the sizing data can be accessed using a general-purpose storage technique.
- If one suspects that an EOP is running out of stack space during execution, it is sufficient to pre-initialize only the lowest few elements in the stack vector, and examine these after a failure.

Terminating the host file server

The host server is a slave process running on the host system, at the same time as the transputer application runs. The top-level process on the root transputer must tell the server when to terminate, and thereby return control to the host operating system. This can be done to the `iserver` as follows :

```

#include "hostio.inc"
#USE    "hostio.lib"

so.exit(fs, ts, sps.success)

```

Note: `sps.success` is declared in the `hostio.inc` file.

Re-running the application without reloading

In most cases, it is convenient to be able to re-run a transputer network application without having to reboot the network. This is achieved by using an OCCAM `WHILE TRUE` loop in top-level process on *each* transputer node in the network. Re-run is achieved by invoking the host server without specifying a boot file to load, but retaining all other command-line options.

For example, an outline of the top-level transputer process on the system's root transputer is :

```

WHILE TRUE
  SEQ
    PAR
      ... run application
      ... terminate host server

```

When the server terminate command is sent to the host, the user is aware of return of control to the host operating system. But the transputer network has entered a state of readiness to be re-run.

Only the root transputer in the system requires to terminate the host server.

Process priorities

It is possible to run an EOP at either high or low priority, in exactly the same way as an OCCAM process. Exactly the same constraints and guidelines apply to non-OCCAM processes as for OCCAM processes, in selecting the priority of execution. So, for example, it would be perfectly reasonable to execute a non-OCCAM process at high priority if it performed a lot of communication to other transputers.

The default priority should be to execute at low priority.

While on the subject of process priorities, it should be observed that it is not obvious how best to obtain performance timing information from processes at high priority. For example, supposing one wished to time the interval between two events in an EOP running at high priority. To obtain a good timing resolution, the high priority clock is to be used.

As a kick-off, to read the high priority timer from a low-priority OCCAM process, the following OCCAM code can be used :

```
PRI PAR
  clock ? before
SKIP
```

This assumes a suitably declared **TIMER** for the **clock**. This fragment can be used anywhere within a low-priority OCCAM process to read the high priority timer, and allow meaningful timing measurements to be made.

To signal to the timing measurement mechanism the start and stop for the event under investigation, one method would be for the non-OCCAM process to send a message on a channel, and to use the receipt of the message as a timing reference. For a C EOP, the arrangement might look like this :

```
#define SIGNAL 1
{
  _outword(SIGNAL, out[2]); /** signal before event **/
  ... do the event to be timed
  _outword(SIGNAL, out[2]); /** signal after event **/
}
```

The word **SIGNAL** is sent as an indication of the start and stop of the event within the process. Some corresponding OCCAM for this arrangement would be :

```
PRI PAR
  PAR -- high
  ... run non-occam process being timed at high priority
  SEQ
  signal ? any
  clock ? before -- immediately before event

  signal ? any
  clock ? after -- immediately after event
  ... run rest of code at low priority
```

The problem with this arrangement is one of scheduling. Once the high priority EOP has sent its signal message, and the OCCAM has read the message using **signal ? any**, the OCCAM will deschedule (due to a communication) and the EOP will re-schedule until it sends the terminate signal. Only at this point, will the clock be read *corresponding to the first signalling*. If the EOP happens to signal the event completion at the end of the EOP process itself, the **before** and **after** timings will be read almost immediately

consecutively, giving results of 1 or 2 microseconds regardless of the event one intended to time. This is clearly not robust.

The correct way to make timings of involving high-priority processes in this way is to *force* a lock-step synchronization between the event being timed and the timing process. This can easily be achieved by incorporating a simple acknowledge protocol between the OCCAM and the C. The OCCAM now uses an **ack** channel, which can be read by the EOP.

```
PRI PAR
PAR -- high
... run non-occam process being timed at high priority
SEQ
  signal ? any
  clock ? started -- immediately after startup
  ack ! frig      -- essential acknowledge

  signal ? any
  clock ? stopping -- immediately before stopping
  ack ! frig      -- essential acknowledge
... run rest of code at low priority
```

The C fragment (run at high priority) then becomes :

```
#define SIGNAL 1
{
  int ack;
  _outword(SIGNAL, out[2]);      /** signal before event **/
  _inmess(in[2], &ack, 4);      /** ack lockstep sync **/

  ... do the event to be timed

  _outword(SIGNAL, out[2]);      /** signal after event **/
  _inmess(in[2], &ack, 4);      /** ack lockstep sync **/
}
```

Another way to force lock-step, but without using an extra acknowledge channel, is to have the EOP send a pair of signals for each event to be recorded. The OCCAM process reads the timer between the two signals from the EOP, thereby forcing lock-step.

11.6 D705B debugging guidelines

This section discusses some concepts which are useful in connection with using the toolset debugger supplied with the D705B.

11.6.1 Problems with conventional debugging techniques

In a parallel system, one cannot use conventional debugging techniques. For example, the traditional strategy of causing screen or file output to represent the passing of a specific point in the program cannot be used with reliability. This is because other processes executing in parallel may cause processor resource to be deflected from causing the anticipated output.

Furthermore, in a multiple process system, there is generally only one (user) process (the *root process*) which is directly connected to the host file server. This is true in systems containing one or several transputers, and in mixed-language systems too. This can often present problems when one is attempting to debug a system of processes, because of the hassle of having time-stepped status information routed from processes deep in a network to the screen or to a file for later perusal.

11.6.2 Error mode considerations

The error mode employed in compilation of harnesses is important. The scientific-language compilers have no concept of the OCCAM compiler's error modes. With the D705B, however, the error mode adopted by an EOP is that of its harness (the EOP). The following discussion concerns debugging opportunities in a customer's software development and production phases.

- **Development phase**

To debug correctly and effectively, one requires three things; the HALT error mode harness, a halt-on-error bootstrap, and the host file server's `/se` error test directive.

For the development environment, the use of error mode HALT is advised. This will cause a halt-on-error bootstrap to be employed automatically by the bootstrap tool, and will allow the debugger to be used for post-mortem debugging and correct location to the source line causing the error. This error mode *must* be used in conjunction with a halt-on-error bootstrap *and* the host server's `/se` error test directive to allow correct and effective debugging of scientific-language systems.

Note that the requirement of HALT mode for debugging purposes requires that *all* OCCAM referenced in the system must be compiled in HALT mode.

- **Production phase**

For a customer's production software, the use of error modes UNDEFINED and UNIVERSAL is recommended. This will allow the fastest execution due to the minimal run-time checking of the OCCAM parts in the system, and also avoid unnecessary termination due to the transputer's error flag becoming set. All the scientific-language compiler range can cause the transputer's error flag to be set during exceptional circumstances in normal processing (a performance-driven feature). Only by adopting these error modes will a non halt-on-error bootstrap be prepended automatically to the linked object file by the bootstrap tool.

However, such conditions do not permit correct error-location by the debugger. This is because running a system with the server invoked with the `/se` option is not sufficient to stop the actual *transputer* process, even although the `iserver` will terminate immediately. The transputer process will continue to execute until it has to communicate with the server — and then stop of necessity because the server has. This would cause the debugger to locate to the wrong line of source.

11.6.3 Run-time debugging aids

When debugging a scientific-language system, it is frequently useful to be able to halt the transputer if a specific assertion is found to be true at *run-time*. One way to achieve this is to use a simple function, written using the C compiler's assembler-insert mode, to set the transputer's error flag depending on the value of a parameter passed to the function. For example,

```
void assert(test)
int *test;
{
    if (*test)
        asm {
            sethalterr;
            testerr;
            seterr;
        };
}
```

The function first selects the processors's halt-on-error mode, using the `sethalterr` instruction. This allows the function to be used in systems that have not been used with a halt-on-error bootstrap. It then tests the error flag, with a view to clearing it. The `seterr` instruction sets the error flag unconditionally. It is necessary to clear the error flag and then set it for the halt-on-error mode to cause the transputer to halt. If the error flag was already set then the introduction of the halt-on-error mode would *not* halt the processor if

the halt-on-error mode was not indigenous to the current execution. Although the error flag is not preserved during normal process descheduling, there are no deschedulable instructions in this function, so if the test is true then the transputer will halt. (The error flag *is* preserved when a high priority process interrupts a low priority process) [5].

This binary object of this function can be linked in with any scientific-language system compilation units, as shown previously in this document. It is called with a single integer reference parameter. A reference parameter has been used to accommodate the FORTRAN reference parameter passing mechanism. A C caller would use the reference & operator for the assertion test parameter. A Pascal caller would require visibility of the function using this technique :

```
IMPORT procedure assert ALIAS 'assert' (VAR test: INTEGER);
```

If the parameter references a value that is not zero, the transputer will halt dead, allowing the debugger to locate to this line of source. The procedure call invocation trace-back facility can be used to find out where the function was called from in that specific instance, and thereby determine the current state of the program under examination.

11.6.4 Debugging processes that are not connected to the host server

This section discusses a simple-to-implement post-mortem technique for debugging and examining the status of any or all processes in a multiple processor environment, and is equally effective for any of the supported transputer source languages. It allows strategic information capture and storage, which the debugger can examine following program execution.

Overview of technique

The technique relies upon the use of a circular buffer, preferably one per transputer in the system, which is connected to each process on the same transputer that one wishes to monitor. The technique is for the user to embed debug information in each process required, and to have this information captured in time sequence from all active processes. The programmer can then use the D705B toolset's debugger to examine the contents of the circular buffer. Providing one outputs sensible messages to the buffer, one can gain an overview of the status of not only each individual process in the system, but also of all the processes on that transputer as they synchronize and interact together. An implementation of this is shown in Figure 11.12. The EOPs in the diagram consist of the EOP plus supporting OCCam processes.

One could have a monitor process for each EOP, or one that accepted input from many EOPs. Both cases are illustrated. Monitor 1 is shown as handling EOPs 1, 2, and 3 (EOP 3 is the root process). This monitor is being used to examine the timing interactions between the EOPs on transputer 1. Unless a timing interaction was being investigated, it would not normally be useful to have the root process (EOP 3) contributing to a message buffer because of the ease of accessing the host's display or filestore.

Monitors 2 and 3 (for EOPs 4 and 5) are shown as servicing debug data from only one EOP each. In this case, it's because the EOPs in question are on different transputers. But it's also useful for examining lots of trace points within an EOP but without concern as to how the execution of the EOP is related to the rest of the system. The debug data in question is received on a channel allocated and controlled by the programmer's message preparation routines in the EOP.

Implementation detail

There are two parts to consider in the implementation. First, the data storage buffer, of which one is required per transputer. Secondly, the debug message preparation code, used by each process in the system.

- **The data storage buffer**

Each transputer in a network will possess a top-level OCCam harness which describes how all processes on that transputer interact with each other (and with those on other transputers). To implement the debug monitor system, an additional process called `circ.buf` is added to the OCCam. The process defines and manages a circular **BYTE** buffer, and accepts input messages from any number of connected processes. Each message from any process has the *same* format,

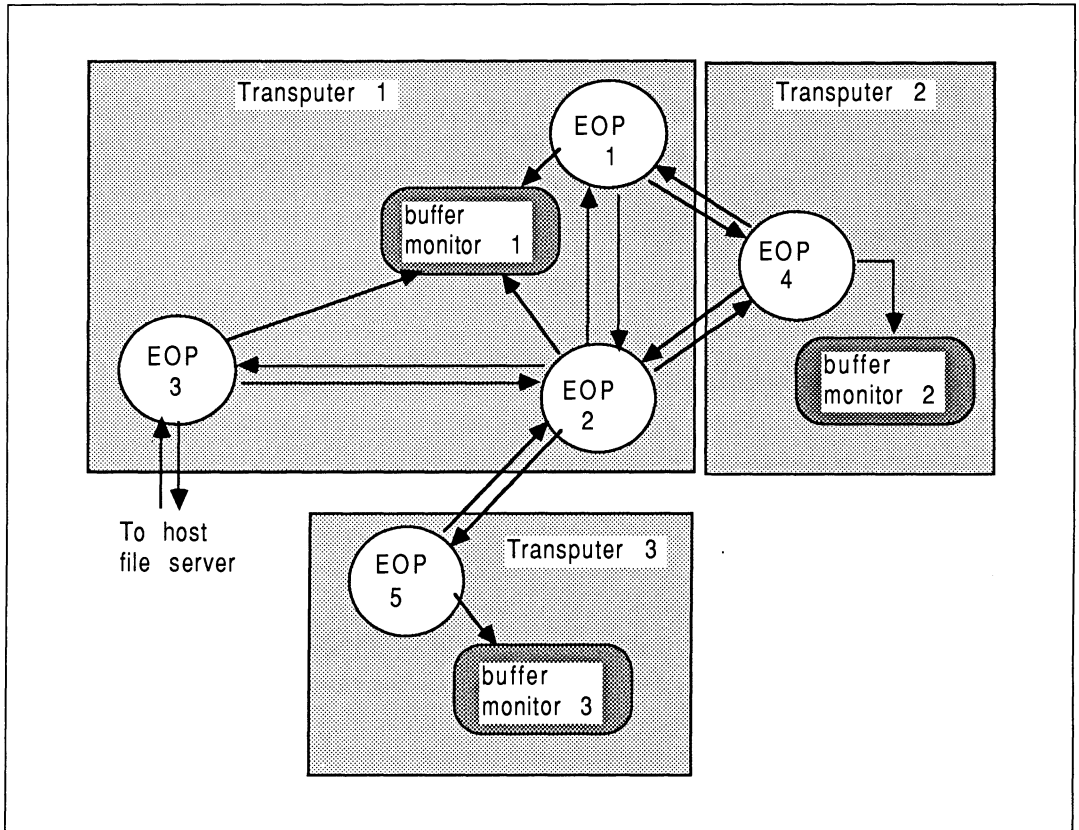


Figure 11.12 General purpose information capture and storage for post-mortem debugging

allowing the buffer to be general-purpose. An OCCAM protocol called **p.MESSAGE** is used to enforce the communication format. The format consists of an integer identifying the number of bytes of message about to be received from that process, followed by a byte vector of that size.

One possible implementation of the buffer manager is shown below :

```

PROC circ.buff (CHAN OF INT RootHasTerminated,
               []CHAN OF p.MESSAGE UserDebug)

  VAL INT BUFFSIZE IS 2000 : -- BYTES of buffer
  [BUFFSIZE]BYTE buffer :
  [100]BYTE last.message :
  INT pointer, process, mess.length :
  BOOL going :

  PROC insert.message (VAL []BYTE message)
    SEQ i = 0 FOR SIZE message
    SEQ
      buffer[pointer] := message[i]
      pointer := ((pointer + 1) REM BUFFSIZE)
    :

  SEQ
    pointer := 0
    going := TRUE
    WHILE going
      PRI ALT
        -- Terminate input command
        INT any :
        RootHasTerminated ? any
        SEQ
          going := FALSE
          insert.message("!! Normal termination !!")
          STOP
        -- Normal message storage
        ALT i = 0 FOR SIZE UserDebug
          UserDebug[i] ?
            mess.length::[last.message
                          FROM 0 FOR mess.length]
          SEQ
            ... insert ID number of process into buffer
            insert.message ([last.message
                            FROM 0 FOR mess.length])
        :

```

The parameters to the `circ.buffer` consist of a channel which is used to terminate the buffer manager, and an array of channels which are used to receive debug input messages in the correct format from an arbitrary number of processes. The termination of the buffer manager is considered next.

• Termination considerations

In a multiple process system, the user's design will provide for one process that should terminate last. For example, on the root transputer, the root process communicating with the host file server should terminate last, because there should be nothing useful happening afterwards. But here, the buffer manager should never terminate, although it can be signalled of the root process's shut-down.

This is so that the debugger can easily examine the workspace it used.

```

WHILE TRUE
  SEQ
    CHAN OF ANY RootStopped :
    [2]CHAN OF ANY UserDebug :
    ... other channel definitions
  PAR
    SEQ
      ... run root non-occam process
      RootStopped ! 1 -- stop debug buffer manager
      ... terminate iserver

      ... run second non-occam process
      ... run third non-occam process
      circ.buff(RootStopped, UserDebug)

```

- **Message preparation code**

Each process requiring debugging must have the capability to prepare meaningful messages in the correct format; an integer length followed by a byte vector. This is simple to achieve in OCCAM. Also, because all the INMOS scientific-language systems provide message-passing functions, this can be easily achieved in other languages too.

Without going into too much detail, some general principles should be expounded. Firstly, the channel used for the outputting of debug messages should be exclusively used for that purpose. Secondly, a designated group of functions / procedures should have exclusive use of this channel, ensuring that all output is of the correct format.

As an example of this, consider a C process that one wishes to debug using the circular buffer technique. The following C function can be used to output a message in the correct format to the circular buffer manager process. This function conforms to the `p.MESSAGE` protocol used by the OCCAM buffer manager.

```

debug(message)
char *message;
{
    int len;

    len = strlen(message);
    _outword(len, out[DEBUG_OUT_CHAN]);
    _outmess(out[DEBUG_OUT_CHAN], message, len);
}

```

Once can write a simple suite of functions to package integers and floating point data into strings for outputting to the buffer in the correct format. Once written, these routines can be used from processes written in other languages, in any system one can mention.

- **Using the debugger**

The debugger is run on the transputer network after running the memory dumper program. It can express the address at which the message buffer is stored in memory, and the current value of the buffer pointer. Returning to the debugger's monitor page allows one to do an ASCII dump of the memory map starting at this address. One can then read out all the debug messages that were captured during the program's running.

This technique can be used to perform post-mortem debugging on an arbitrarily complex transputer network. The technique and its component tools are universally applicable and totally general purpose once written.

What to do if you don't have a debugger

Buy the D705B!

Alternatively, for use in environments such as the D705A or Parallel C/FORTRAN where no debugger is provided, the above technique is still important. Instead of having the debugger investigate the contents of the data storage buffers, the application itself dumps the buffer contents to the screen. For transputers other than the root transputer, the buffer contents must be routed back to the host using a simple protocol like the one used to place messages in the buffer in the first place.

If you happen to own additional PC's and transputer boards or link adapter cards, then it is possible to have more than one non-OCCAM process linked with the full run-time library. This would permit "probing" of a troublesome process not directly connected to the host server on the main host computer, because auxiliary output can be observed using the other PC. It's a long shot but it might just work ! Try it

11.7 Using the D705B occam-2 toolset

This chapter describes some worked examples using the D705B OCCAM toolset. It is presented in a tutorial fashion, and can be read in front of a computer while doing the examples. Following an overview of makefiles, a twin EOP system using one, then two transputers is shown. Use of the D705B libraries is also explored. A technique for sharing code modules amongst EOPs is demonstrated, in the context of the debugging monitoring buffer.

Refer to section 11.8 for a checklist on what has to be set-up to allow the D705B to be used correctly.

This chapter discusses topics in the context of the PC-based D705B. Toolset operation would be exactly the same in any of the toolset platforms (but it should be remembered that the switch-character is a ' - ' in UNIX-based toolsets). The EOPs can be compiled and linked on a PC, then transferred to a Sun-3 or VAX for integration with a toolset on that machine. There would be no change in tool operation or procedure.

11.7.1 About makefiles

Makefiles specify how all the different parts of a system depend on each other. A makefile allows a tool, called **make**, to perform the minimum number of operations to correctly update a system following changes in any number of parts of that system. The D705B toolset uses makefiles in this way.

The format of commands in a makefile is significant, in terms of spaces and tab characters. So, for example, the following two lines in a makefile

```
dualharn.c4x:  dualharn.l4x dualharn.t4x
              $(LINK) /f dualharn.l4x $(LINKOPT)
```

indicate that the file **dualharn.c4x** depends on two files called **dualharn.l4x** and **dualharn.t4x**. When the **make** tool processes the makefile, if any of the files to the right of the colon are more recent than the one to the left of the colon, then it will execute the following command **\$(LINK) /f dualharn.l4x \$(LINKOPT)**. The directives involving dollar signs and round braces are macros, which are defined at the top of the makefile. These are optional, but have been used here to allow the programmer to easily change the boot commands and options to all the toolset tools. In this example, the command will run the linker if the compiled OCCAM (**.t4x**) or the linker command input file (**.l4x**) is more recent than the output file from the linker (**.c4x**).

The D705B tool **imakef** generates makefile descriptions of a systems' interdependencies. This will be shown in the examples.

11.7.2 Two communicating EOPs on one transputer

Suppose we have two EOPs and we wish them to execute concurrently on the same transputer. Using the D705B OCCAM toolset, each EOP can be enclosed by a simple harness, with a top-level harness describing

how the EOPs interconnect.

In order not to obscure the details of operating the toolset and of constructing the supporting OCCAM, the EOPs will be deliberately trivial. Of the two processes, the "root" process will display messages on the screen, consisting of data sent to it from the "remote" process which has a Type 2 interface. The remote process is only remote in the sense that it is not directly communicating with the host file server, and consequently is linked with the standalone run-time libraries — it has a Type 3 procedure interface.

Operations overview

Firstly, the non-OCCAM source is compiled and linked with the necessary run-time library support. At the same time, OCCAM development can proceed. The OCCAM harness will reference each EOP using the `#IMPORT` directive. The HALT execution mode is used to facilitate debugging during development. A makefile description of the system is built using the `imakef` tool. Once the non-OCCAM code has been linked, the system can be built.

Consider in turn the two EOPs.

The root EOP

This process outputs messages to the screen, representing data sent to it from the remote process. A tagged protocol is used, allowing firstly a sequence of integer numbers to be received, followed by a sequence of character information. In C, this could be implemented as follows.

- The source

```
#include <chanio.h>

#define OUT_CHAN    2
#define IN_CHAN    2

#define STOP        0
#define NUMBERS     1
#define LETTERS     2

typedef int CHAN;

main (argc, argv, envp, in, inlen, out, outlen)
    char *argv[], *envp[];
    int argc, inlen, outlen;
    CHAN *in[], *out[];
{
    int value, count, size, total, tag = 0;

    printf("\nHow many items in the first group ? ");
    scanf("%d",&total);
    _outword(total, out[OUT_CHAN]);

    printf("\nSTARTED\n");
    _inmess(in[IN_CHAN], &tag, 1);
    while (tag != STOP)
    {
        if (tag == NUMBERS)
        {
            _inmess(in[IN_CHAN], &value, 4);
            printf("%d\n", value);
        }
        else if (tag == LETTERS)
        {
            _inmess(in[IN_CHAN], &size, 4);
            for (count = 0; count < size; count++)
```



```

        {
            _inmess(in[IN_CHAN], &value, 4);
            printf("%c\n", value);
        }
        _inmess(in[IN_CHAN], &tag, 1);
    }
    printf("FINISHED\n");
}

```

Notice that this process is expecting to receive its messages on channel two (see pre-processor definition for **channel**) of the previously-described input vector of channels to the process. This communication is facilitated by the additional arguments shown to **main()**. When we write the supporting OCCam, we must ensure the remote process and this process are correctly connected up together – this is not a compile-time issue for the scientific-language process.

- **Building it**

For an IMS T414, assuming this source is stored in a file called **cprog1.c**, this is compiled using the command **t4c cprog1**. The object binary must then be linked with the full standard run-time library and the Type 2 interface :

```

ilink NonOcc1=procent.c4h cprog1.bin crt1t4.bin /o
nonoccl.c4h

```

This creates a linked file called **nonoccl.c4h**, which is **#IMPORTed** into the OCCam EOP harness, and instanced using the identifier **NonOcc1**. From this stage onwards, the linked compilation unit is treated as a normal OCCam PROC, and the reference to “nonocc” is simply intended as a reminder of where the mixed-language components fit into the scenario. The **c4h** filename extension indicates that the file contains linked object code, compiled for a T414 in HALT error mode.

Notice the EOP run-time library **crt1t4.bin** does not have a directory path specified, even although it is not in the same directory. This is due to the library path-searching mechanism in the D705B³, which uses a DOS environment variable **ISEARCH**, and could be set up as follows :

```

ISEARCH=c:\itools\libs\;
c:\itools\interf\;
c:\tc1v3\;
c:\tp1v2\;
c:\tf1v1\;

```

The directories specified in **ISEARCH** are searched to locate files that are not in the directory in which the tool was invoked.

³The ISEARCH is not a true DOS path specification, because it is textually prepended to filenames while searching the list of directories. Notice the trailing backslashes, for instance.

The remote EOP

This process sends messages to the root EOP described above. The tagged protocol used in this process must conform to that expected by the recipient process. Again in C, one possible implementation is as follows :

- The source

```
#include <chanio.h>

#define OUT_CHAN  2
#define IN_CHAN   2

#define STOP      0
#define NUMBERS   1
#define LETTERS   2

typedef int CHAN;

main (argc, argv, envp, in, inlen, out, outlen)
  char *argv[], *envp[];
  int  argc, inlen, outlen;
  CHAN *in[], *out[];
{
  int current, total;
  _inmess(in[IN_CHAN], &total, 4);
  for (current = 1; current <= total; current++)
  {
    _outbyte(NUMBERS, out[OUT_CHAN]);
    _outword(current, out[OUT_CHAN]);
  }

  _outbyte(LETTERS, out[OUT_CHAN]);
  _outword(3, out[OUT_CHAN]);
  for (current = 65; current <= 67; current++)
    _outword(current, out[OUT_CHAN]);

  _outbyte(STOP, out[OUT_CHAN]);
}
```

Notice that this C source has a `main()` body — every separate C process has `main()` as its entry point, regardless of its position within a transputer network. Again, this process will send its data on word two of the output vector of channel pointers supplied to the process. The OCCAM to be described is responsible for ensuring the channel connections intended by the user are in fact correctly established.

- Building it

If this source is stored in file `cprog2.c`, then it can be compiled for the T414 using the command `t4c cprog2`. Since this process uses *only* channel message passing to communicate (ie, it doesn't use `printf`), it will be linked with the reduced standalone run-time library and a Type 3 interface :

```
ilink NonOcc2=procentc.t4h cprog2.bin sacrtlt4.bin /o
nonocc2.c4h
```

This creates a linked file `nonocc2.c4h` which is `#IMPORTed` into the OCCAM EOP harness, and instanced using the identifier `NonOcc2`.

- Building both EOPs with a makefile

It is advisable to write a separate makefile for the non-OCCAM software. It is impractical for the D705B to create a makefile for non-OCCAM software, because of the required information concerning module compilation and link requirements etc.

A suitable makefile for the two EOPs in this example would be as follows :

```
# makefile for non-occam software

all:      nonoccl.c4h nonocc2.c4h

nonoccl.c4h:      cprog1.bin
                  ilink NonOcc1=procent.c4h cprog1.bin
                  crtlt4.bin /o nonoccl.c4h

nonocc2.c4h:      cprog2.bin
                  ilink NonOcc2=procentc.t4h cprog2.bin
                  sacrtlt4.bin /o nonocc2.c4h

cprog1.bin:      cprog1.c
                  t4c cprog1

cprog2.bin:      cprog2.c
                  t4c cprog2
```

If this makefile was called **nonocc**, then to build the non-OCCAM components of the system automatically, type **make -f nonocc**.

In the above two C routines, it is important that the communications protocol used by the two partners is consistent. In other words, the protocol tags used must correspond at each end of the communications channel. The best way to guarantee this is to place the communication tag constants into a **#include** file, and reference this file in both C sources. This technique is also appropriate for communicating Pascal partners. Unfortunately, the V1.1 FORTRAN compiler does not support a source textual file inclusion mechanism, because this is not part of the ANSI standard. Parallel FORTRAN does support source file inclusion.

It is not advised that the actual communications channel indexes (**OUT_CHAN** and **IN_CHAN** above) are placed in a **#include** file shared between the EOPs, because in most cases the communications channel indexes for both EOPS, and indeed, in either direction, will be different. But all source components of any one EOP should share this data.

The occam bits

The OCCAM required consists of a harness for each EOP, and a top-level interconnection. Assume the source is stored in the file **dualharn.occ** :

- The source

```
#INCLUDE "hostio.inc"
PROC NonOcc.entry (CHAN OF SP from.link, to.link, [ ]INT free.memory)

-- IMPORTS are nonoccl.c4h, nonocc2.c4h
#USE "hostio.lib"

PROC p.NonOcc1 (CHAN OF SP fs, ts,
               CHAN OF ANY from.outside, to.outside)

[3]INT in.NonOcc :
[3]INT out.NonOcc :
SEQ
  LOAD.INPUT.CHANNEL (in.NonOcc [2], from.outside)
  LOAD.OUTPUT.CHANNEL (out.NonOcc[2], to.outside)
```

```

#IMPORT "nonocc1.c4h"
[1]INT dummy.ws :
[5000]INT work.space :
-- type 2 interface
NonOcc1(fs, ts, 1, work.space, dummy.ws,
        in.NonOcc, out.NonOcc)
:

PROC p.NonOcc2 (CHAN OF ANY from.outside, to.outside)

[3]INT in.NonOcc :
[3]INT out.NonOcc :
SEQ
LOAD.INPUT.CHANNEL (in.NonOcc [2], from.outside)
LOAD.OUTPUT.CHANNEL(out.NonOcc[2], to.outside)

#IMPORT "nonocc2.c4h"
[1]INT dummy.ws :
[5000]INT work.space :
-- type 3 interface
NonOcc2(1, work.space, dummy.ws, in.NonOcc, out.NonOcc)
:

WHILE TRUE
SEQ
CHAN OF ANY OneToTwo, TwoToOne :
PAR
-----
p.NonOcc1 (from.link, to.link, TwoToOne, OneToTwo)
-----
p.NonOcc2 (OneToTwo, TwoToOne)
-----

so.exit (from.link, to.link, sps.success)
:

```

- Building it

The D705B **imakef** utility controls the sequence of commands required to create your executable application. In this case, it will control the OCCAM compiler, the linker, and the bootstrap tool. To run the **imakef** utility, specify the type of file you want to build. Here, we want to build a bootable file for a T414, in HALT mode. This implies a **.b4h** file extension. So, we issue the command :

```
imakef dualharn.b4h /i
```

This creates a file called **dualharn**, which lists the file dependencies and tool invocation commands, and a file called **dualharn.l4h**, which is a control file for the linker.

The **dualharn** file contains the following :

```

LIBRARIAN=ilibr
OCCAM=occam
LINK=ilink
CONFIG=iconf
ADDBOOT=iboot
LIBOPT=
OCCOPT=
LINKOPT=
CONFOPT=
BOOTOPT=

dualharn.b4h:  dualharn.c4h
               $(ADDBOOT) dualharn.c4h $(BOOTOPT)

dualharn.c4h:  dualharn.l4h dualharn.t4h
               $(LINK) /f dualharn.l4h $(LINKOPT)

dualharn.t4h:  dualharn.occ nonocc1.c4h nonocc2.c4h
               \itools\libs\process.lib
               \itools\libs\hostio.lib
               $(OCCAM) dualharn /t4 /h $(OCCOPT)

```

This file is a makefile.

The linker command input file created, **dualharn.l4h**, contains this :

```

dualharn.t4h
c:\itools\libs\hostio.lib
c:\itools\libs\convert.lib
nonocc1.c4h
nonocc2.c4h
OCCAMBH.LIB

```

This file indicates the list of binary objects to be linked. The **OCCAMBH.LIB** file is the OCCam compiler library, which is automatically included by the makefile generator. The reference to **convert.lib** exists because the hostio library has a library usage file associated with it. The programmer need not be aware of this, except when manually linking components together.

To initiate the build, type **make -f dualharn**. This results in the following commands being run automatically :

Command	Takes as input	Makes as output
occam dualharn /t4/h	.occ	.t4h
ilink /f dualharn.l4h	Files listed in .l4h	.c4h
iboot dualharn.c4h	.c4h	.b4h

This results in **dualharn.b4h**, a bootable file. The table does not show the creation of supplemental files.

Running the program

To boot the program, use the **iserver** :

```
iserver /sb dualharn.b4h /se
```

The result will be a short sequence of numbers and characters on the screen, depending on the user input. The server will then terminate and control will return to the host operating system prompt. The following display is observed when the number "3" is specified at run-time :

```

STARTED
1
2
3
A
B
C
FINISHED

```

The application can be re-run without reloading by calling the `iserver` directly with only the “serve link” `/ss` option. This is a direct consequence of the `WHILE TRUE` construct in the OCCaM harness.

Rebuilding

To rebuild the system, following editing changes, is simple. If changes were made to any of the non-OCCaM programs, then the makefile for them must be used to re-generate new `.c%%` linked files. Then, all the necessary OCCaM components are updated using the makefile produced by the D705B `imakef` tool. For example, following changes to a system that did not affect or introduce more file dependencies, the following two commands are sufficient to reconstruct the system :

```

make -f nonocc
make -f dualharn

```

It is only necessary to alter the makefiles or re-run the `imakef` tool if there is any alteration to the file dependencies of the system.

Re-implementation of the EOPs

Suppose one wished to re-implement the root EOP, referenced with the identifier `NonOcc1`, in a different language. Previously, a C implementation was shown. To implement a functional equivalent in Pascal, for example, to slot into the existing framework, one could do the following :

```

program root (input, output);
$include '\tplv2\channels.inc'

const
  OutChannel = 2;
  InChannel  = 2;

  Stop      = 0;
  Numbers   = 1;
  Letters   = 2;

var
  tag : char;
  value, count, total : integer;

begin
  write('How many items in the first group ? ');
  readln(total);
  outmess(OutChannel, total, 4);
  writeln(' STARTED');
  inmess(InChannel, tag, 1);
  while (tag <> chr(Stop)) do
    begin
      if (tag = chr(Numbers)) then
        begin
          inmess(InChannel, value, 4);
          writeln(value);
        end
    end
end

```

```

else if (tag = chr(Letters)) then
  begin
    inmess(InChannel, value, 4);
    for count := 1 to value do
      begin
        inmess(InChannel, value, 4);
        writeln(chr(value));
      end;
    end;
    inmess(InChannel, tag, 1);
  end;
writeln('FINISHED');
end.

```

This Pascal source is functionally equivalent to the C function described in earlier sections. Place this source in the file called `pasprog1.pas`, and adjust the `nonocc` makefile as follows :

```

nonocc1.c4h:  pasprog1.bin
              ilink NonOcc1=procent.c4h pasprog1.bin
              prtlt4.bin /o pasprog1.c4h

pasprog1.bin: pasprog1.pas
              t4p pasprog1 /x

```

The `/x` option permits the Pascal compiler to make use of the message-passing extensions to the standard language definition to which the compiler conforms.

Run `make` on both system makefiles, and reload the program as before. It's as simple as that. No changes are necessary to the OCCAM.

Similarly, to re-implement the remote EOP in FORTRAN :

```

PARAMETER (IOUTCHAN=2, INCHAN=2)
PARAMETER (ISTOP=0, NUMBERS=1, LETTERS=2)
INTEGER VALUE, TOTAL
VALUE = 1
CALL CHANINMESSAGE (2, TOTAL, 4)
DO 10 I = 1, TOTAL
  CALL CHANOUTBYTE (NUMBERS, IOUTCHAN)
  CALL CHANOUTWORD (VALUE, IOUTCHAN)
10  VALUE = VALUE + 1
  CALL CHANOUTBYTE (LETTERS, IOUTCHAN)
  CALL CHANOUTWORD (3, IOUTCHAN)
  VALUE = 65
  DO 20 I = 1, 3
    CALL CHANOUTWORD (VALUE, IOUTCHAN)
20  VALUE = VALUE + 1
  CALL CHANOUTBYTE (ISTOP, IOUTCHAN)
STOP
END

```

Place the source in file `fprog2.f77`, and adjust the `nonocc` makefile as follows :

```

nonocc2.c4h:  fprog2.bin
              ilink NonOcc2=procentf.t4h fprog2.bin safrtlt4.bin
              /o fprog2.c4h

fprog2.bin:  fprog2.f77
             t4f fprog2

```

The reduced run-time library is used for this FORTRAN process, in the same way as for the C and Pascal examples. Again, there is no need to alter or re-compile the other non-OCCAM process. To rebuild the

system, simply **make** the two makefiles. The program behaviour is exactly the same.

11.7.3 Two communicating EOPs on two transputers

This section describes how to use the D705B to build a multi-processor system, using the EOPS of the previous examples. The EOPS will be used unchanged, one on each transputer. The EOP harnesses **p.NonOcc1** and **p.NonOcc2** will be used unchanged — total portability! Each transputer will require a top-level OCCAM process to connect to the EOPs. In addition, a network configuration description will be required.

Let the top-level OCCAM processes for each transputer be called **mainharn.occ** and **auxharn.occ** :

Source of **mainharn.occ** :

```
#INCLUDE "hostio.inc"
PROC NonOcc.root (CHAN OF SP from.link, to.link,
                 CHAN OF ANY OneToTwo, TwoToOne)

  #USE "hostio.lib"

  ... PROC p.NonOcc1 from previous example

  WHILE TRUE
    SEQ
    -----
    p.NonOcc1 (from.link, to.link, TwoToOne, OneToTwo)
    -----

    so.exit (from.link, to.link, sps.success)
  :
```

The source of **auxharn.occ** :

```
PROC NonOcc.remote (CHAN OF ANY OneToTwo, TwoToOne)

  ... PROC p.NonOcc2 from previous example

  WHILE TRUE
    -----
    p.NonOcc2 (OneToTwo, TwoToOne)
    -----
  :
```


The network configuration description is stored in a file with a `.pgm` extension, say `multcon.pgm` :

```
#USE "mainharn.c4h"
#USE "auxharn.c4h"

VAL links.out IS [0, 1, 2, 3] :
VAL links.in  IS [4, 5, 6, 7] :

CHAN OF ANY main.to.aux, aux.to.main :

PLACED PAR
  PROCESSOR 0 T4
    CHAN OF SP from.link, to.link :
    PLACE from.link AT links.in [0] :
    PLACE to.link   AT links.out[0] :
    PLACE aux.to.main AT links.in [2] :
    PLACE main.to.aux AT links.out[2] :
    NonOcc.root (from.link, to.link,
                 main.to.aux, aux.to.main)

  PROCESSOR 1 T4
    PLACE main.to.aux AT links.in [1] :
    PLACE aux.to.main AT links.out[1] :
    NonOcc.remote (main.to.aux, aux.to.main)
```

Assuming that the `nonocc` makefile is used to create the linked `.c%` EOPs, then all that has to be done is to use the `imakef` tool to construct dependency information. This is done (only once) as follows :

```
imakef multcon.btl /i
```

A makefile `multcon` is created, and linker control files for each processor, `mainharn.14h` and `auxharn.14h`. To build and re-build the system, the two makefiles are used in sequence :

```
make -f nonocc
make -f multcon
```

If the entire system has to be built, the operations invoked by the second make are as follows :

Command	Takes as input	Makes as output
<code>occam mainharn /t4/h</code>	<code>.occ</code>	<code>.t4h</code>
<code>ilink /f mainharn.14h</code>	Files listed in <code>.14h</code>	<code>.c4h</code>
<code>occam auxharn /t4/h</code>	<code>.occ</code>	<code>.t4h</code>
<code>ilink /f auxharn.14h</code>	Files listed in <code>.14h</code>	<code>.c4h</code>
<code>iconf multcon</code>	<code>multcon.pgm</code>	<code>.btl</code>

This results in a file called `multcon.btl`, suitable for booting a transputer network down a link :

```
iserver /sb multcon.btl /se
```

The program behaviour is exactly the same as before, except it now runs on two transputers. Neither the EOPs or their OCCAM harnesses had to be altered. And it can still be re-run without reloading.

Note that because vanilla OCCAM can be used at configuration level, it would have been possible to dispense with the `NonOcc.remote` procedure, and directly called `p.NonOcc2` from configuration level :

```
... rest of configuration file
PROCESSOR 1 T4
  PLACE main.to.aux AT links.in [1] :
  PLACE aux.to.main AT links.out[1] :
  WHILE TRUE
    p.NonOcc2 (main.to.aux, aux.to.main)
```

There's always more than one way to do anything!

11.7.4 Using the debugger with the twin EOP twin transputer system

Supposing an error occurs during the execution of the twin transputer system, described above. The transputers will stop dead because HALT mode has been used. The `iserver` will stop if the `/se` option was used at run-time. In this situation, it is necessary to make a "coredump" of the root processor so that the debugger can load onto it. The command to make the coredump (of, say, 100000 bytes into a file called `multcon.dmp`) and load the debugger, are :

```
coredump multcon 100000 multcon.bt1
```

This command makes use of the coredumper and the debugger, in the following way :

```
idump multcon 100000
idebug multcon.bt1 /r multcon
```

The debugger will then locate to the line causing the error; even if this occurred during execution of a non-OCCAM process. To be fully effective, the EOP harnesses should all be compiled in HALT mode, and the server would be run with the `/se` error test option.

11.7.5 Placing the EOPs in a library

It is possible to place EOPs in libraries, which can then be used by OCCAM processes. For example, the compiled and linked EOPs in the previous section can be placed in a library. The library mechanism is very flexible, because libraries can refer to items in other libraries, and the different modules in a library are all selectively loadable by the linker depending on the satisfaction of outstanding external references, the processor type, and error modes.

It is not recommended to use the `imakef` tool to generate a makefile for libraries containing non-OCCAM components. This is because the `imakef` tool assumes the existence of OCCAM source for all binary object components, and it would create a lot of un-necessary make information if it were used.

As an example, both `nonocc1.c4h` and `nonocc2.c4h` will be placed in a library called `EOPlib.lib`. Both `mainharn.occ` and `auxharn.occ` will reference `EOPlib`, but because `mainharn` only references the EOP called `NonOcc1`, then only the module containing that item will be linked with `mainharn`. The same is true of `auxharn`, but for `NonOcc2`.

The procedure here is to call the librarian directly :

```
ilibr nonocc1.c4h nonocc2.c4h /o EOPlib.lib
```

Using the `ilist` binary lister tool, you can check the library contents :

```
ilist EOPlib.lib /e
```

This will give the following display :

Entry Pt	Module Name	No	TT	EM	Offset	Wspace	Vspace
NonOcc1	ill:nonocc1.c4h	0	414	H	508	143	1474
NonOcc2	ill:nonocc2.c4h	1	414	H	0	21	0

This indicates that the library `EOPlib.lib` contains two modules (either of which can be independently loaded into an application), both suitable for execution on a T414. Module 0 has an entry point name of `NonOcc1`, derived from the contents of file `nonocc1.c4h`, and Module 1 has an entry point name of `NonOcc2`, derived from the contents of file `nonocc2.c4h`. The OCCAM source of `mainharn.occ` and `auxharn.occ` is modified to reference the library by using the command `#USE "EOPlib.lib"`.

11.7.6 Sharing code amongst EOPs in a system

Share and Enjoy. It is possible for the EOPs in a transputer system to Share and Enjoy some common code in certain circumstances. The requirements are that the EOPs reside on the same transputer, and the code that they share is implemented in OCCAM. This provision allows for the standard OCCAM libraries to be shared between any number of EOPs, in addition to the programmer's own OCCAMPROCs.

The example to be given is that of the circular buffer debugging technique, shown in C in Section 11.6.4. Three EOPs run on the root transputer. They all require to contribute messages to the buffer to examine timing relationships during execution. The buffer manager is implemented in OCCAM and uses OCCAM library procedures; and the code is to be shared by all EOPs.

Consider firstly the non-OCCAM components in the system.

The EOPs

Each C EOP would have the following stub called `debug`, which would reference a shared OCCAM procedure called `debugocc`. To avoid passing more parameters than necessary, the `debugocc` procedure will be compiled without separate vectorspace (by using the `/v` option). However, the size of the message being passed must be included as an explicit parameter in the C (it's a hidden parameter in the OCCAM). Each EOP could use a different channel for outputting the diagnostic debug messages on.

```
#define DEBUG_OUT_CHAN    3

debug(message)
char *message;
{
    debugocc (out[DEBUG_OUT_CHAN], message, strlen(message));
}
```

Because each EOP has to share the OCCAM PROC called `debugocc`, the makefile for the EOPs must allow the linker to leave unresolved external references (the `/u` option). For example, an extract from the makefile used to generate the EOP interface for the C program `cprog1` :

```
all.c8x:      cprog1.bin
              ilink EOP1=procent.c8x cprog1.bin crt1t8.bin
              /o all.c8x /u

cprog1.bin:   cprog1.c
              t8c cprog1
```

The shared OCCAM code

The `debugocc` PROC is filed in `or.occ`, perhaps like this :

```
PROC debugocc (INT dummy, CHAN OF ANY debug.chan,
               [ ]BYTE string)
  -- There is a hidden parm for the size of string
  SEQ
    debug.chan ! SIZE string
    debug.chan ! [string FROM 0 FOR SIZE string]
  :
```

The relevant part of a makefile to generate the compiled `.t8x` output is :

```
or.t8x: or.occ
        occam or /t8/e/i/v/x
```

Notice it's compiled *without* separate vectorspace, in UNIVERSAL error mode. However, the main OCCAM harness for the processor is to be compiled in HALT mode. Code compiled for HALT mode can call code compiled for UNIVERSAL mode, but not the other way round. It could have been compiled in HALT mode.

If the main OCCAM harness for the whole processor is called **debugv.occ**, then the linker control file **debugv.18h** might look like this :

```
debugv.t8h
c:\itools\libs\hostio.lib
c:\itools\libs\convert.lib
or.t8x
a11.c8x
a12.c8x
a13.c8x
OCCAM8H.LIB
```

To show that only one copy of the OCCAM procedure **debugocc** has been linked in to the system, the linker generates a link map automatically. This is filed in **debugv.m8h**, and looks like this :

```
SC debugv.t8h 0 643
SC a13.c8x 644 3875
SC a12.c8x 3876 7303
SC a11.c8x 7304 45955
SC or.t8x 45956 45999
LIB c:\itools\libs\convert.lib (3) 46000 46131
LIB c:\itools\libs\hostio.lib (18) 46132 46207
```

The link map shows that the placement of compilation units is not related to the ordering of items in the linker control file **debugv.18h**. The linker is free to arbitrarily re-order items. If it is especially important to have certain compilation units placed low down in memory (in the hope of placing them on-chip), then the linker symbol optimization facility can be used.

Linker symbol optimization

To use the linker symbol optimization facility, the programmer specifies the symbol names which have to be "optimized". The optimization takes the form of placing the specified symbols at the start of the items to be linked. The hope is that the modules at the start of the list will be placed on on-chip RAM, and thereby execute the most rapidly — effective use of on-chip RAM is what symbol optimization is all about. If the modules happen not to fall on-chip, then there is no tangible benefit in having them optimized using this technique. See Section 11.7.6 for guidelines on calculating where the tools place specific modules.

The linker's **/q** parameter specifies the symbols to be optimized, all of which are taken as equal priority for optimization. The **/q** directive can be placed inside the linker control file **debugv.18h**, or on the command line. So, including the directive

```
/q (debugocc, EOP1)
```

in the linker control file **debugv.18h** would place **or.t8x** (entrypoint symbol **debugocc**) at the head of the link map, and **a11.c8x** (entrypoint symbol **EOP1**) immediately after it. The rest of the modules to be linked will follow in the same order as before. Check them by examining the **debugv.m8h** link map :

```
SC or.t8h 0 43
SC a11.c8x 44 38695
SC debugv.t8h 38696 39339
SC a13.c8x 39340 42571
SC a12.c8x 42572 45999
LIB c:\itools\libs\convert.lib (3) 46000 46131
LIB c:\itools\libs\hostio.lib (18) 46132 46207
```

The default is for the linker to optimize the symbols **REAL32OP** and **REAL32OPERR**, if they are used by the program.

With respect to the treatment of symbol optimization, the ordering of module placement is the same as the order in which the component objects are listed in the linker input specification (the **debugv.18h** file). So, if it were vital that the **a11.c8x** module were placed *before* the **or.t8x** module, the correct approach

would be to edit the linker control file `debugv.18h` and ensure that `a11.c8x` is placed before `or.t8x`. Re-ordering the symbol entrypoints in the `/q` directive would have no effect.

If one of the library modules had to be “optimized”, and only the module number (shown in parentheses in the `debugv.18h` link map) is known, then the `ilist` utility should be used on the library in question. The specific module numbers can be listed with the `ilist's /s ()` option, and the use of `/e` ensures that the entrypoint symbols are listed. One can then have the required module optimized by the linker.

Calculating where specific modules are placed

It can be useful to be able to calculate where specific code modules are placed on a transputer. For example, by careful use of the linker symbol optimization facility, one can endeavour to place critical modules in on-chip RAM. In some transputer boards, the external memory is stratified in performance terms (eg, the INMOS B404 TRAM module) with a certain amount of low-down fast static RAM, topped up with slower dynamic RAM. Even in these situations, code module placement can affect execution speed.

It is possible to calculate where any specific module is placed in the transputer's memory map. This breaks down into two parts. The first task is to determine where the start of the code area is. The second task is to determine the offset of the module of interest from the code start area. Consider each in turn :

- **Calculating the code block start**

The code start area is most easily calculated by not calculating anything at all — if you see what I mean. Use the debugger to find out where the code start area is, on any transputer in your network.

Assuming you have just run your single processor application, say `debugv.bh8`, then the debugger would be used like this :

```
idump debugv 100000
idebug debugv.b8h /r debugv
```

This causes the core dumper to store 100000 bytes of data from the root processor to a file called `debugv.dmp`. The debugger then loads into the root processor, and refers to the `debugv` core-dump file for information about the root processor.

Alternatively, the program descriptor (with reference to the previous examples its `debugv.d8h`) can be used. Here's the descriptor `debugv.d8h` from the previous example :

```
Occam Toolset Make Bootable V1.0
PROCESSOR 0 0 T800 1
SC 0 46208
SCNAME debugv.c8h
CODE 20 0 1164 68124 46208 0
```

The relevant line is the line beginning with the word `CODE`. Without going into too much detail, this descriptor says there is one T800 transputer in the system, and that there is one linked `SC` filed as `debugv.c8h`. The relevant fields in the `CODE` specification are the first one, 20, which indicates the number of bytes used for configuration information, and the third field, 1164, which indicates the number of bytes used by the OCCAM scalar workspace. Recall from earlier sections and diagrams that the code block (mixed OCCAM and otherwise) is placed immediately after the scalar workspace block.

In fact, to be strictly accurate at this point, the code block begins above the configuration code, which is above the scalar workspace, which is above `MemStart`. `MemStart` takes the value 112 (#70) on a T800 or T425, 72 (#48) on a T414, and 36 (#24) on the T2 family. So, the actual calculation for the position of the start of code is `MemStart` plus scalar workspace plus configuration code. In this example, $112 + 1164 + 20 = 1196$, or #510. This is the same answer as the debugger would

give in its memory map display :

```

                Memory Map
Workspace       : #80000070 - #800004BF ( 1164 )
Configuration code : #800004FC - #8000050F ( 20 )
Program Body    : #80000510 - #8000B98F ( 46k)
Vectorspace     : #8000B990 - #8001C3AB ( 67k)

Total memory usage : 115628 bytes (113k)

```

Notice that the total memory usage shown by the debugger tells you how large a core dump file you should have used!

As an aside, the other numbers in the descriptor identify the vectorspace requirements (68124 bytes, or 17031 words), and the code size of the `debugv.c8h` linked module, 46208 bytes.

If the information option had been used on the bootstrap tool, the vectorspace size is shown in words (17031). The scalar workspace requirement is also shown in words, 282. The `ilist` utility will confirm these two numbers. However, an “extra” nine words are included in the scalar workspace by the configuration operation, as far as the descriptor and debugger are concerned⁴. Hence, $282 + 9 = 291$ words (or 1164 bytes on a T800) are reserved for scalar workspace once the code has been rendered bootable.

• Calculating the module offset position

This is simply a matter of tracing backwards, starting with the module requiring position location, and finding out all the things that are linked in with it. Each time the module is linked with other object code, the linker will produce a link map (in a `.m**` file). The position of the module in that particular linked unit can be observed from the byte position addresses shown in the link map. Simply add together the module offsets shown in each `.m**` file, to determine the *total* offset of the module from the start of the code.

Alternatively, one can trace the module position forwards from the top-level linked unit which has the bootstrap prepended, through all the intermediate linkings to the module under investigation.

The absolute module position is then determined by adding the module offset address (from code start) to the code start address.

Using on-chip RAM effectively

Knowing the start and end addresses of critical modules, (the byte sizes of each module can be derived from the `.m**` files), it is apparent whether part or all of the module is in on-chip RAM.

For performance reasons, it may be important to fit a particular combination of modules in on-chip RAM. With reference to the above example, the size of the scalar workspace is such that the program body starts at 1296 (#510), but the T800 on-chip RAM extends to only 4095 (#FFF). This leaves $4095 - 1296 = 2799$ bytes (#AEF) of on-chip RAM for the code.

Following the use of the linker symbol optimization in the previous example, the first two items loaded are :

```

SC or.t8h 0 43
SC all.c8x 44 38695

```

The `or.t8x` is an indivisibly loadable unit. However, the `all.c8x` comprizes other parts. There is a

⁴This information is highly specific to the current D705B implementation, and is not guaranteed to remain the same for all releases of the D705B tools. The appropriate product documentation should always be consulted.

corresponding linker map file for this, called **a11.m8x**. The first parts of this file are listed below :

```

SC procent.c8x 0 9571
LIB crt1t8.bin (59) 9572 11467
LIB crt1t8.bin (39) 11468 12327
LIB crt1t8.bin (77) 12328 14255

```

The actual C object file **cprog1.bin** appears much further down the list. Since only 2799 bytes of code are available on-chip, clearly the actual user-code is not placed on-chip. If it were vital that **cprog1.bin** was on-chip, it must be brought to the head of the link list. To force **cprog1.bin** to the head of the link list, the **/q (EOP1)** directive would be included in the linker control specification for building **a11.c8x**.

This is clearly a trivial example, but the methodology is applicable to any size of problem. You can make programs execute faster. What a great plan ! I'm excited to be a part of it ! Let's do it !!!

11.7.7 Hints and tips

This section includes a few tips on how to get the best out of the D705B toolset. These sections are also relevant to any other toolset platform.

Library usage guidelines

These notes address some library usage issues.

- Many complete EOPs can be put into a library, and access to all of them is available with only one **#USE** directive. However, the makefile generator tool **imakef** will generate incorrect makefiles if it finds a **.lib** library build file for non-OCCAM material. This is because it will assume OCCAM source exists for for everything, which is not true for EOPs.
- It is not possible to mix source and object code in the same file. A consequence of this is that files of OCCAM source **VAL** declarations and **PROTOCOL** specifications cannot be put into a library. Rather, they must be filed separately and accessed by **#INCLUDE**, with a recommended filename extension of **.inc**.
- Object hex output from the scientific-language compilers cannot be placed in libraries. Convert it to binary using the scientific-language linker, and then put this in a D705B library.
- Separately compiled functions / procedures belonging to an EOP can be placed in a library. The object fragments of an EOP cannot be linked until *all* the component binary objects are available.
- Use of the generic processor classes in libraries allows compact libraries to be created from OCCAM source that support a range of processors. If it is necessary to produce a library supporting all 32-bit processor types, then attempt to compile for a processor class TA. If this is not possible due to the nature of the code, then class TB and T8 together, or class TC and T4 together, cover all processor types. Failing this, the library must contain T4, T5, and T8 compilation units to offer the same support.

Remember though, that use of generic processor classes causes restrictions in the instructions that can be used. For example, TA cannot do floating point.

- Careful use of the OCCAM compiler's error modes can contribute towards compact libraries. In totally OCCAM systems, the HALT mode is advised for testing and debugging purposes. If a routine is known to be correct, or has severe performance constraints, the UNIVERSAL error mode in a library allows any type of compilation unit to access the routine. A corollary of this item and the previous one is that **.tax** compilation units can be used by the greatest range of processor types and error modes.
- Most libraries are built from compiled **.t%%** components. In situations where it is required to reduce the number of entry-points to a library, or the number of unresolved external references, linked **.c%%** components could be used as an alternative to inserting the other necessary **.t%%** files.

If OCCAM source to be placed in a library uses only textual references to other OCCAM files (using **#INCLUDE**), then there are no external references from the compiled unit. Therefore, the compiled output (`.t%`) would be placed in a library.

If OCCAM source references compiled items with **#USE** or **#IMPORT**, then this means that the compilation unit `.t%` possesses unresolved external references. If the `.t%` file were inserted to a library, any programs using the library would also have to use the libraries that satisfy the external references of this one. To remove this condition, the compilation unit can be linked to resolve its external references, and the resultant `.c%` unit placed in the library. This makes the unit more “portable”, in the sense that it can now be used without the other libraries. An equivalent approach would involve inserting the other `.t%` units in the same library, and not using linked `.c%` units at all — this is the approach used to build the D705B toolset libraries.

One cannot place compiled OCCAM that references object code with the **#SC** directive in a library.

- *Don't* use the OCCAM compiler's **#SC** directive! It is only supported for compatibility reasons with the TDS compiler. The restrictions with **#SC** on linking position and the impossibility of inclusion in a library are easily avoided. Instead, use the **#USE** directive — and compile as normal. This makes the compiler treat the item as a library. Remember that a library can be a single object file, so it is simply a case of changing occurrences of **#SC** to **#USE**, for advantage to be taken of the library features available. The advantages of using the **#USE** directive over the **#SC** are numerous, and include selective loading, arbitrary placement opportunity at link-time, and only one copy of the code is linked in no matter how often it is **#USED** (on one processor).

General usage guidelines

This section contains generally useful advice for using the D705B toolset.

- In general, don't explicitly specify absolute or relative directory locations in OCCAM directives to access other files. This compromises the OCCAM source-level portability amongst the other toolset platforms. The **ISEARCH** path mechanism should be used instead, as it essentially offers a machine-independent “logical naming” facility as in the INMOS TDS. If it is necessary to use a machine-dependent form of file specification, then stick firmly to either relative directories or absolute directories — don't mix or your source becomes very confusing and non-portable.
- Because the linker cannot create directly a bootable file, there is the overhead of having to store a `.c%` file which represents the entire process for the transputer, but which is not bootable. If you are running low on disk space, and building large applications, you can delete the auxiliary symbol maps for each linked compilation unit (`.s%`) and also the `.c%` files *after* adding the bootstrap. This is because the `.c%` files are no longer required (unless you ask the debugger for a code/memory comparison). Don't delete any `.t%` and `.m%` files because the debugger uses these. Better still, use the linkers' `/s` symbol table disable option.
- The D705B linker attempts to resolve external references unless given a `/u` option to disable this. It also requires that an entry point for the binary object be provided. The practical implication of this is that using the D705B linker, non-OCCAM code can only be pre-linked as a complete entity, with the compiled OCCAM interface code included in the specification of files to be linked (such as `procent%.t%`).
- Because each **#IMPORT**ed EOP has always been linked with the same standard OCCAM interface code, then a means of speeding up system re-generation time is possible (assuming that the OCCAM interface code is not altered). If changes have been made to non-OCCAM components in a system, but not to OCCAM components, then it is *not* strictly necessary to recompile any OCCAM. It is, however, necessary to link and bootstrap the code as before.

The **imakef** tool would arrange for an OCCAM recompilation if it detected the linked `.c%` file referenced in a **#IMPORT** were more recent than the OCCAM source referencing it. To prevent this, it is necessary to tweak the system makefile. Manually remove the dependency information for the OCCAM concerning the **#IMPORT**ed `.c%` files. Then, arrange that the makefile for the non-OCCAM parts will delete the `.c%` files which comprize the compiled OCCAM and the **#IMPORT**ed

stuff. This ensures that once the non-OCCAM parts have been rebuilt as necessary, the OCCAM compiler will not be invoked on account of the **procent** interfacing routines — but it will still be invoked if any OCCAM has changed. Remember, it is *always* necessary to re-link and bootstrap the application following an editing change.

- If the debugger's Network Dump option is used with a single transputer system, then if the application happened to use the **free.memory** buffer (for run-time stack and heap storage, or for other OCCAM buffer allocations), this memory will not be saved to disk. Only memory allocated from within the OCCAM harnesses is stored in this case.
- To use the debugger with a (single transputer) application which uses **free.memory**, then to ensure the used portion of this memory is core-dumped for the debugger to use, two approaches can be taken. Either set **IBOARDSIZE** smaller than it is (to 200000 bytes, say, instead of ten times that). This means that used memory is lower down in the memory map, so a single core-dump of reasonable size can be taken. Alternatively, use the core-dump tool to dump several blocks of memory. The **idump** can accommodate a list of up to ten *start / size* byte pairs.

11.8 Some useful checklists

11.8.1 Setting things up for the D705B

There are a few things to set up before you proceed :

- Ensure the D705B toolset search path **ISEARCH** is set up for the toolset *and* non-occam compiler directories, and ensure it has trailing backslashes for each component path.

Reminder : With MS-DOS, spaces in setting up environment variables *are significant*. It is very easy (and not obvious what's happened if you do it) to set up an environment variable called "space"-**ISEARCH!**

- Ensure that the DOS environment variable **IBOARDSIZE** is set to the size of the transputer board, eg, #200000 for a 2 Mbyte board. If you think you've set up this environment variable, and the **iserver** terminates with a fatal error, then you may have set up an environment variable called "space"-**IBOARDSIZE** (which is not useful).
- The **ITERM** environment variable is used by the debugger, and must point to a valid .ITM file, such as IBMPC.ITM in the appropriate directory.
- The CONFIG.SYS file must install the ANSI.SYS device driver, otherwise the debugger and simulator will not correctly draw the screen.
- You may need to increase the number of FILES and BUFFERS in your CONFIG.SYS file, to something like 20 or 30. This requirement may arise if a tool making use of a lot of files / buffers (such as **imakef**) is unable to proceed for any obvious reason (like disk space exhausted, file write-protected, file doesn't exist, search paths not correct etc).

11.8.2 What to do if a multiple EOP system won't run (on one transputer)

This section is a checklist for when a multiple EOP system doesn't execute correctly. It assumes that the multi-EOP system compiles, links, and loads OK, but won't run. The checklist is applicable to any multi-process D705B application, and is listed in order of check-ability.

- Ensure that each EOP has been linked with the correct type of OCCAM interface code. Generally, there will be one type 2 EOP and the remainder will be Type 3 EOPs. Remember the interfaces are different depending on the language of implementation of the EOP.
- Check that the (Type 2) root EOP has been linked with the full run-time library. All other (Type 3) EOPs should be linked with the standalone libraries (unless they use the Type 3 *stub* interface).

- Are the message-passing functions being given the correct type of arguments ? In particular, note that the C functions `_inmess` and `_outmess` take *addresses* as the second parameter! (*Not* constants).
- Ensure the EOP OCCaM harness has the correct `LOAD . INPUT . CHANNEL` and `LOAD . OUTPUT . CHANNEL` commands, and isn't using any of the reserved scientific-language communications channels. In summary, elements 0 and 1 of both channel vectors `in` and `out` are reserved for an EOP using the full run-time libraries, and element 0 of both vectors is reserved for an EOP linked with the standalone run-time library.
- Has sufficient workspace been reserved in the OCCaM instantiation of each EOP?

If an EOP uses two workspaces (`flag` is 0), then a minimum of 400 words for `ws1` stack, and 4000 words for `ws2` heap is recommended.

If an EOP uses one workspaces (`flag` is 1), then a minimum of 4000 words for `ws1` (all workspace) is recommended. In this case, `ws2` can be of size 1.
- Do the channels used within the EOP source actually correspond to the ones the programmer has used in the OCCaM used to interconnect the channels ? In other words, does the EOP harness expect a C EOP to send data on channel `out [2]`, but the C source sends the data on a different channel ?
- Ensure that the EOP source does not explicitly attempt to use hard link addresses with the message passing functions. C EOPs *must* use the elements of the `in` and `out` vectors passed as arguments to `main()`, rather than using `#define` to place channels onto the hardware.
- Ensure that each channel communication pair send and receive the same number of bytes, otherwise partner will jam.

What do you mean it *still* won't run ?

DON'T PANIC !!!

11.8.3 What to do if a multiple EOP system won't run (on many transputers)

Clearly, the first stage is to get the system to run on a *single* transputer first. Don't be too ambitious initially and dive into a multi-processor implementation — make it work with one transputer first.

If you have a system that works on one transputer, but fails to run when configured for several, then following checklist is useful :

- The first thing to check is that all the channels are correctly **PLACEd** onto the hard links.
- Have you declared the root processor *first* in the configuration description file ?
- Is it necessary to establish link connections *before* booting the application (for example, by using the Module Motherboard Software to set up the INMOS C004 link switch on a B008 motherboard).
- Are your processor types in correspondance with those declared in the configuration file ?
- Do you have enough memory on each processor node ? (Check this by getting the configurator to produce a boot map for you. This also lists the code requirements for each processor).
- Are all the link speeds compatible between adjacent processors ? Check the DIP switches on your motherboards.
- Check that all processors are being correctly reset, especially where a hierarchical reset control strategy is being employed, eg, one involving the SubSystem ports.

11.8.4 A summary of performance maximization techniques

This section lists the main three areas for increasing a system's performance, without going into total detail of how to drive all the tools to achieve this.

- Use the tools effectively.
- Use the on-chip RAM effectively.
- Write your software correctly.

There is some obvious overlap between these categories.

Examples of all three categories follow :

- **Use the tools effectively**

The C / Pascal / FORTRAN compiler's `/PCn` option allows the programmer to change the number of bytes allocated for a call to an **extern** function, which is to be patched by the linker. The default is to save 6 bytes, allowing a maximum code image of 16 MBytes. Often, values like `/PC4` (giving 64KBytes) and `/PC5` (giving 1 MByte) can be used to make code smaller and execute faster. The linker will warn if too small a patch size is used, and also informs the programmer of the maximum patch size it used. Legal values are `/PC2` to `/PC8`.

The C compiler's `/S` option can be used to prevent the C compiler converting all floating point operations to **double** precision before evaluating expressions. This is not recommended for applications where high numerical accuracy is required, but is faster.

The D705B toolset linker **ilink** can also be used effectively to minimize code size by sharing OCCam code, even between parallel EOP processes running on the same processor.

The linker can also assist with the effective use of on-chip RAM for critical parts of the code.

- **Use the on-chip RAM effectively**

Ensure that the stack space of compute-intensive parts of the application is placed on-chip.

Use the linker correctly to ensure that the most frequently used functions are loaded on-chip (if possible) — the linker provides maps showing the loading order of component binaries in the final executable image — use them to find out where the code is loaded, allowing for **Memstart** (#70 bytes (112 decimal) on T800 and T425; #48 (decimal 72) on T414), the OCCam workspace (convert to bytes !), and around 20 bytes of reserved configuration info, preceding the code.

It is possible to use **KERNEL.RUN** techniques in association with those discussed previously to *guarantee* certain code will reside on-chip. This is discussed in another technical note.

- **Write your software “correctly”**

Distribute compute-intensive parts across multiple processors.

Always overlap slow i/o (such as communication to the server) with computation.

Use lots of buffers to decouple communication and computation — especially software talking to inter-processor links.

Communicate in one large “chunks” at a time, rather than in several smaller quantities.

Don't use arithmetic that is explicitly *not* 32-bit (on the 32-bit transputers). For example, in **occam**, the **INT16** data type is manipulated much more slowly than 32-bit **INTs** (especially when part of a vector). Pay the storage penalty and reap the benefits of performance! Try to use machine native-wordlength computation.

What more can I say ? Contact Central Applications group with your personal favourites.

11.9 Summary and Conclusions

This document has described some issues connected with developing transputer software using the INMOS scientific-language development systems and the D705B **occam** toolset. Most of the examples shown can be copied verbatim and used as templates in the reader's own projects⁵, using any **occam** toolset on any supported platform.

In addition to fulfilling the requirements of new projects, in any language, these development systems allow *existing* applications to be ported to transputers.

The development systems are thorough and flexible. All support a range of transputers. The D705B offers multiple programmer support, and application compatibility at source and binary levels across a range of development platforms. Transputer software is fast, incrementally upgradable, and portable. Can you afford to be without it ? Inject some life into your application ! Use the Toolset.

11.10 References

- 1 *The Transputer Databook*, INMOS Limited
- 2 *occam-2 Reference Manual*, INMOS Limited, Prentice Hall
- 3 *Some Issues in Scientific-language Application Porting and Farming using transputers*, INMOS Technical Note 53, Andy Hamilton, INMOS Limited, Bristol
- 4 *INMOS Spectrum*, (contains a brief description of INMOS products), INMOS Limited, Bristol
- 5 *Transputer instruction set — A compiler writer's guide*, INMOS Limited, Prentice-Hall.
- 6 *INMOS Parallel C User Guide (V2.00 software)*, INMOS Limited, Bristol
- 7 *INMOS Parallel FORTRAN User Guide (V2.00 software)*, INMOS Limited, Bristol
- 8 *Porting SPICE to the INMOS IMS T800 transputer*, INMOS Technical Note 52, Andy Hamilton and Clive Dyson, INMOS Limited, Bristol
- 9 *Performance Maximization*, INMOS Technical Note 17, Phil Atkin, INMOS Limited, Bristol

⁵Some small print : A set of unsupported example programs discussed in this Technical Note, are available from INMOS by contacting a Field Applications Engineer. Send a disk and we'll send you the examples.



Quality and Reliability

A Quality and Reliability

Systems products are embraced within the INMOS Quality Policy which incorporates specific programmes in the following areas:

- Design in quality
- New product verification phase
- Document Control
- Quality control monitors
- Production soak testing
- Environmental stress cycle
- Reliability testing

All systems products are designed in house using CAD facilities specific to PCB manufacture. These facilities incorporate design simulation and provide production data which helps to reduce design to production problems.

During the product verification phase, the new product is evaluated and its build/test specifications are endorsed.

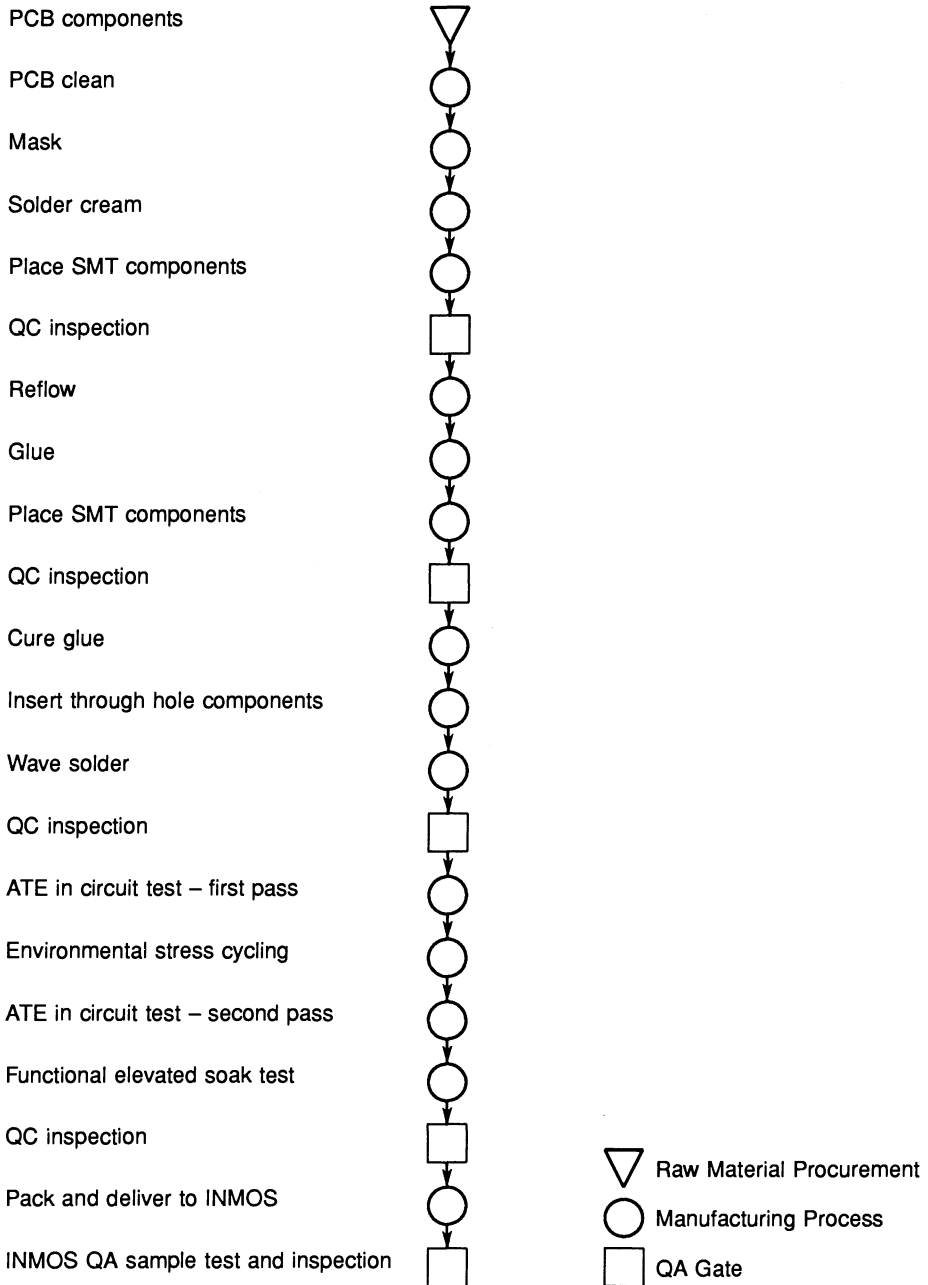
All system products are assembled and tested at INMOS approved assembly houses which conform to the INMOS Quality Program. Quality procedures detail the build specification, production testing and final product status. These procedures are all monitored and controlled by the Document Control Department (DCD).

In circuit automatic testing provides assembly and component analysis. This coupled with an environmental stress cycle and a repeated automatic test procedure provides an effective monitor to the production phase.

An INMOS Quality Assurance sample test evaluates all production batches. At this stage the conformance of the product is confirmed and the test data logged for reference.

Reliability testing is carried out on the major product lines. Samples are taken from standard stock and subjected to life testing.

PRODUCT FLOW





Cables for Board Products

B Cables for board products

The following cable sets are available to complement the INMOS range of board products. Sufficient cables are included with each of the INMOS board products to build the most common configurations. However, where more sophisticated systems are required, it will sometimes be necessary to use additional cables. The table below indicates the number of each cable type included in each of the available cable sets.

Part Number	Cable Reference Code																		
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	
IMS CA01						10	2	1	1										
IMS CA02					8					2									
IMS CA03	10	10	1	1															
IMS CA04				3					1										
IMS CA05												3							
IMS CA06											1								
IMS CA07													1						
IMS CA08														1					
IMS CA09															3	10	10		
IMS CA10																			1

Table B.1 Cable sets

Code Description

A	Short link cable (approx. 0.1m)
B	Standard link cable (approx. 0.5m)
C	Long link cable (approx. 1.0m)
D	Very long link cable (approx. 2.0m)
E	Link jumper (yellow wires)
F	Short reset cable (approx. 0.1m)
G	Standard reset cable (approx. 0.5m)
H	Long reset cable (approx. 1.0m)
I	Very long reset cable (approx. 2.0m)
J	Reset jumper (black wires)
K	DIN 41612 socket to wire wrap tails (with no 'row b' pins)
L	DIN 41612 socket to wire wrap tails (with 'row b' pins)
M	DIN 41612 socket to INMOS link connections
N	DIN 41612 to BNC (video) connectors
O	3-way SIL subsystem pin strip
P	8-way SIL TRAM pin extender strips
Q	8-way SIL TRAM slot pipe jumper
R	Pixel bus terminator module

Table B.2 Cable descriptions

The table below shows how many of each cable type are shipped as standard with INMOS board products. This table is only included to show which items are required should it be necessary to replace the cables for a particular product. Only the cable sets shown in table B.1 are available separately from INMOS. The table below is for reference purposes only. (The cable reference codes below refer to the cables described in table B.2).

Board Product	Cable Reference Code																	
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
IMS B001		2				1					1							
IMS B003	4	4			2	1	1					1						
IMS B004		2			1	1				1								
IMS B005		1					1											
IMS B006	8	4		2	2	1			2			1	1					
IMS B007		2				1					1			1				
IMS B008		2					1								1		10	
IMS B009	3				1	1				1								
IMS B010	2	2			1	1				1						8		
IMS B011			4					1			1							
IMS B012	4	4		2	1	1	1	1				1	1		1		16	
IMS B014		4		2			1		1						1			
IMS B409																		1

Table B.3 Board product cables