Tandem

Ken Shillito

COLLABORATORS			
	TITLE : Tandem		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Ken Shillito	August 8, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Tandem

1.1 Tandem Guide

```
                Tandem.guide   Version 0.28   October 7, 1999    | AmigaGuide  ↩
                   version:
                                                 | You should see a single
© 1999  Ken Shillito    Winchelsea,  Australia   | backslash below this line.
email     shillito@tpg.com.au                    |          \
homepage  www2.tpg.com.au/users/shillito         | else click <Help> button!!


                Introduction - What Tandem Does
                   Beginner's Manual - teach yourself Assembly Language

                Beginner's Guide to Assembly Language
                Reference Manual - assumes basic knowledge of assembly language

                Tandem Editor/Assembler/Debugger - User Manual
                Assembly language programmers will find Tandem to be a powerful,  ↩
                   flexible,
and easy to use programming tool. In fact, Tandem makes assembly language
programs nearly as easy to debug as BASIC programs.

                    Optional Supplement

Tandem comes bound up with tandem.library and Front.i, an easy to program
front end, GUI, and assembly language development environment for the Amiga.


                Front.i and tandem.library
                contains full documentation of these, if you want to use them.  ↩
                   However you
can use all aspects of Tandem without ever using tandem.library and Front.i
```

1.2 Index to Tandem.guide

```
                Tandem Guide - Index
```

SECTION

ENDM

MASK2

SET

EQU

MEXIT

SPC

EQUR

NOLIST

TTL

FAIL

NOOBJ

XDEF

FORMAT

NOPAGE

XREF

Pseudo ops - Extension
 Non standard extensions by some assemblers

ADDSYM

DSEG

OBJFILE

ALIGN

DSTRING

ODD

ASCII

ELSE

ORG

BDBUGARG

ELSEIF

```
OUTPUT

BITSTREAM

ENDIF

PAD

BLK

EVEN

PRINTX

BOPT

EXEOBJ

PSTRING

BSS

FILECOM

PUBLIC

CARGS

FO RS SO

PURE

CMACRO

GLOBAL

QUAD

CODE

IBYTES

REPEAT

CSEG

IDENTIFY

RS SO FO

CSTRING

INCBIN

SMALLDATA

DATA
```

Optional Supplement [for your use if required]

```
 Elementary Routines

Currently Popped Window
 The focus of the action

Custom Requesters
 Adding to the possibilities

Dropdown Menus
 Dropdown menus

File Routines
 Files

Front.i - about
 General intro to Front.i

Front.i - How to use
 How Front.i fits in

Incudes - Tandem includes
 Overview of Tandem includes

Input from a Window
 Managing IDCMP

I/O Routines
 Input - Output

Help, Attaching Guide
 Context sensitive online help

Memory Routines
 Memory

Menu Routines
 Menus made easy

Output text to a window
 Text, strings &c

Primitive Routines
 Down to the basics

Rendering Routines
 Rendering see also
Other Rendering

Requester Routines
 Requesters

Slider Routines
 Slider rendering & monitoring

String Routines
 Strings
```

```
TLbad
TLhexasc16
TLprefs
TLreqmuclr
TLunbusy

TLbusy
TLhexasc
TLprogdir
TLreqmuset
TLwcheck

TLbutmon
TLinput
TLprogress
TLreqoff
TLwclose

TLbutprt
TLkeyboard
TLpublic
TLreqon
TLwfront

TLbutstr
TLmget
TLputilbm
TLreqredi
TLwindow

TLbuttxt
TLmultiline
TLreadfile
TLreqshow
TLwindow0

TLchip
TLnewfont
TLreqarea
TLresize
TLwpoll

TLclosefile
TLnm
TLreqbev
TLscreen
TLwpop

TLdata
TLoffmenu
TLreqchek
TLslider
TLwritefile

TLdropdown
TLonmenu
TLreqchoose
```

```
TLslimon
TLwscroll

TLellipse
TLopenread
TLreqcls
TLstra0
TLwslof

TLerror
TLopenwrite
TLreqcolor
TLstrbuf
TLwsub

TLfloat
TLoutput
TLreqedit
TLstring
TLwupdate
```

1.3 helpnode

```
Help


Abbreviations

In this manual:

  sc   means "source code"     }  Remember!!!
  mc   means "machine code"    }  remember!!!


AmigaGuide Reader

Here is an exclamation mark:         !
Here is a backslash:                 \
Here is another exclamation mark:    !

You should see a single backslash character between the two exclamation
marks above. If you don't, then you have an obsolete version of the
AmigaGuide reader, and this AmigaGuide won't display backslashes
properly. That will make the section on MACRO's (and quite a few other
things) rather incomprehensible.

The solution: upagrade your Amiga to AmigaDOS release 3.1

If you're doing programming, you really need the latest version of the
operating system. As I write this, OS3.5 is promised, but not yet out.
```

1.4 manual

The Tandem Editor/Assembler/Debugger - User Manual

Important: the following abbreviations are used throughout this manual

  sc for scource code
  mc for machine code

1.5 intro

Introduction to Tandem

Tandem is an integrated Editor/Assembler/Debugger for the Amiga, with a revolutionary BASIC-like user interface, that will make assembly language as easy to debug as BASIC programs!

The assembler is much faster than the other common assemblers on the market. Tandem assembles direct to memory, and the debugger refers to source code, and can do immediate mode instructions, just like in BASIC debugging.

All the Amiga OS3.1 .i header files are instantly available in pre-assembled form, and your programs never need linking!! There are many other great features also. Check out
Capabilities of Tandem
 for more on Tandem.

After you use Tandem, you'll HATE other editor/assemblers.

Do you Hate Reading Manuals?

Yes, of course you do! But, PLEASE, at least read this:

Basic Concepts
and then this:

INCLUDEs and IncAll.i
Sample Programs

You can load & run the following sample programs, to get a feel for using Tandem:

Teaching/21.asm thru Teaching/72.asm

(When you click "Edit sc" on Tandem's main window, and then select "Load" in the menu, you'll see the "Teaching" directory. Skip numbers 1.asm to 20.asm, and load, assemble & run all from 21.asm onwards).

1.6 gettg

The Basics - Installing, Hardware &c.

Hardware Requirements

Installing Tandem

Getting Started

Using Tandem from the Workbench

Copyright Notice and Disclaimer

The Scope of this Manual

The Tandem Directory and Files

1.7 hardware

Hardware Requirements

Tandem may be run on any Amiga with Workbench 2.04+

Although Tandem can assemble 68030 machine code, and MMU and 68881/2 FPU, it only needs a 68000 computer (e.g. an Amiga 500) to run Tandem.

Tandem is memory hungry, and in practice if you are going to go into assembly language programming, you really ought to have the following minimum setup:

```
 - An Amiga 1200 with accelerator and 68882 FPU and 68030
 - a printer
 - a hard disk drive
 - several megabytes of fast memory, and 2 megabytes of chip memory

But, as I say, any Amiga will run Tandem, even a 1 Meg Amiga 500, as long as
it has Workbench 2.04+
```

1.8 install

```
                Installing Tandem

To install Tandem, just drag and drop the icon of the drawer Tandem is in.

(See
                Tandem's Supporting Files
                 for a list of files that must
be in Tandem's directory).

e.g. Suppose you find Tandem in a public domain CD. You would simply open
say the hard disk "work" window, and drag and drop the Tandem drawer from the
CD's window to the "work" window. If the CD has to be un-archived, you
would un-archive Tandem to Work:

(To un-install Tandem, simply click its icon, and select Delete in the work-
bench menu. If you have ever saved prefs, also delete ENVARC:Tandem)

Tandem Preferences

You can save preferences from within Tandem, which creates a directory:

   ENVARC:Tandem

in accord with normal Amiga usage. Tandem has built-in preferences, and many
users will find no need to set preferences, so for them ENVARC:Tandem will
never be created.
```

1.9 starting

```
                Getting Started

Tandem is run from the Shell (CLI), by double clicking the Tandem.exe
icon.

Tandem takes about 3 seconds (from hard disk) to start itself, since it must
load several data files.

Tandem can open its three windows either on the workbench (default public
screen) or on a private screen. You will see on Tandem's main window a

                Prefs
```

button, where you can specify where you want Tandem
to place its windows.

You will then see Tandem's own screen, with some buttons at the top of a
window covering it, and some preliminary information. This is your operating
environment.

It is also (theoretically) possible to run Tandem from the workbench. See

Starting Tandem from the Workbench
for notes about the startup code required if you intend to debug a ↩
program
while Tandem starts up under the workbench. To run Tandem from the
workbench, select "Show all Files" in the workbench menu, and click the
Tandem icon.

1.10 wbnch

Starting Tandem from the Workbench

Normally, it is best to start Tandem from the CLI. However it is also
possible to start Tandem from the workbench. But if you do, then if you
intend to debug, there is a special problem: that is, that your program will
operate under Tandem's process structure. This means, that your program
must not wait for the startup message (or reply to it), or it will
wait forever, since Tandem has itself already received the startup message.

Therefore, in your startup code, you must comment out the receiving of a
startup message & the replying to it, when debugging, and only remove the
comments when saving the assembled version. Almost all programs are written
to operate under either, by sensing whether under the CLI or workbench, so
in that case simply run Tandem under the CLI.

For more information about writing a program front-end, see:

Writing a Front End

1.11 copyright

Copyright Notice

n.b. It is permitted to distribute Tandem on freely distributable PD disks.
But Tandem must NOT be distributed as part of a commercial program, without
my permission.

Disclaimer

Obviously, I cannot be responsible for any consequences of the use of
Tandem. I supply it as freeware, with no explicit or implied warranties
as to its quality or freedom from defects or suitability for use. I have
put Tandem through an extremely lengthy and detailed debugging process,
and so far as I have been able to discover, it is free of bugs.

1.12 thisman

The Scope of this Manual

This manual is written at a level where some assembly language knowledge
is assumed. The other section of this manual:

Beginner's Guide to Assembly Language
will help you to learn the basics of assembly language, and the ↩
Amiga's
internal operating system.

Tandem is an excellent tool for learning assembly language programming.
In my opinion, people should learn assembler before they learn C language
programming. In the Tandem directory is a sub-directory called "Teaching"
containing sample assembly language files which can be used along with
the "Beginner's Guide" (as above) for interactive learning.

1.13 disk

The Tandem Directory and Files

Tandem must reside in a directory along with other files. The directory
would normally be itself entitled "Tandem". Here is a description of the
various files which the Tandem directory must contain:

```
  /Tandem            /Tandem.info           (drawer)
     Tandem
     Tandem.exe      Tandem.exe.info        (project IconX)
     Tandem.guide    Tandem.guide.info      (project MultiView)
     Multiline.guide Multiline.guide.info   (project Multiview)
     Read First      Read First.info        (project MultiView)
     tandem.library
     logo.iff

     Jottings                               (drawer)
        (various doc files)
     Teachings                              (drawer)
```

```
      1.asm to 72.asm     20.i
   Includes                                (drawer)
      IncAll.i
      Tandem.i
      Front.i
   Support                                 (drawer)
      incall.consts
      ssxref.consts
      opcode.consts
      ssxref_make.asm
      Tandem.FD
      tanlib.i
   Projects                                (drawer)
      Asmfiles                             (drawer)
            test00.asm to test18.asm
   TLDemo         TLDemo.info              (drawer)
      21 to 72
      TLdemo.guide    TLdemo.guide.info
      Run 23  Run24  Run31  Run42
```

If you save prefs from within Tandem, ENVARC:tandem.prefs will exist.

You can do the following (optional) CLI commands if you want to:

```
   copy    tandem/tandem.library  to  LIBS:tandem.library
   delete  tandem/tandem.library
```

You must have the Amiga Developer CD (published by Schatzruhe) to use Tandem, and preferably the 2.04 RKM's, published by Addison Wesley.

```
Tandem          Tandem itself. Obviously must be present.
Tandem.guide    Tandem.guide is what you're reading.
tandem.library  Used by Tandem for its user interface. Must be
                        present. May optionally be put in LIBS:
Tandem.exe      Used for running Tandem under the CLI
```

Here are the sub-directories the Tandem directory should contain:

```
TLdemo          A demo of tandem.library's capabilities
Support         Data files used by Tandem. Must be present.
                        (ssxref.consts, incall.consts, opcode.consts)
Projects        Use this dir for your own projects.
Jottings        Use this dir for your
                Jottings
                .
                        Has some programs for testing Tandem's assemler.
Includes        For .i files. See
                Include
                .
                        (IncAll.i, Front.i, Tandem.i)
Teaching        See
                Beginner's Guide
```

.

1.14 capab

```
Capabilities and Limitations of Tandem

1. Tandem is an editor-assembler-debugger of 68030 machine code programs
   for the Amiga computer, along with FPU 68881/2 and MMU instructions.
   (not 68040 or 68060).

2. Tandem differs from other assemblers in that:

    (i) it always assembles directly into memory.
   (ii) it normally uses no object files, but produces loadable machine
        code files directly without an intermediate linker.
  (iii) it keeps source code in memory where it can be edited, loaded and
        saved, just as one does in BASIC programming.
   (iv) debugging commands refer directly to the source code (known as
        "symbolic debugging"), and you can refer to labels and line numbers
        and view source code lines, just as easily as debugging a BASIC
        program (easier, once you get practiced at it!).
    (vi) Tandem assembles and links in a small fraction of the time taken by
        most other assemblers.
  (vii) Tandem has built-in all the Amiga INCLUDE files and XREF's in
        pre-assembled form; this means that your programs can reference
        all the symbols in the Amiga OS3.1 INCLUDE files, but they
        assemble instantly!

I will now give a general description of Tandem's capabilities, before
proceeding to the nitty gritty of how to use Tandem in practice.

1. Program parameters.

   Tandem allows great flexibility by the user in setting and adjusting
   program parameters. All Tandem's memory buffers may be independently
   adjusted in size (Most users will never need to do so).

2. Source code.  (i.e. assembly language in ASCII)

   In this manual, source code is abbreviated as "sc"

   Source code is kept in memory, as in BASIC programming. Tandem contains
   facilities for editing and re-assembly of source code, etc. from within
   Tandem itself.

   Tandem's source code syntax rules are almost entirely compatible with
   the traditional Metacomco assembler from Amiga's early days. Source code
   can be edited, typed in, loaded and saved interactively, just like in
   BASIC programming. Most of the extensions to assembly language that
   assemblers such as Devpac, Barfly and A68k have added, are also supported
   by Tandem.

3. Object code.

   Tandem can produce object code if required, but in practice object code
```

is rarely needed, since Tandem resolves all _LVO XREF statements at assembly time.

4. The process with other assemblers:

edit source code - run assembler - run linker

is compressed into one step only with Tandem - there is generally no linking step required. After using Tandem, you'll HATE other assemblers.

5. Machine code, debugging.

In this manual, machine code is always abbreviated as "mc"

Tandem assembles directly into memory, from where Tandem can also issue the usual debugging commands. Machine code can also be saved. Tandem is not only an assembler and linker, but a symbolic dubugger. And a very easy to use symbolic debugger at that.

6. Assembly.

Tandem uses a super-fast assembly algorithm (which is admittedly memory hungry) making assembly & linking 3 or 4 times faster. If you are going to do program development on an Amiga 1200, you really need to get an accelerator, and memory expansion, and perhaps a 68882.

An amalgamated file made from all the Amiga OS3.1 INCLUDE files - i.e. an enormous file of over 600,000 bytes, with many thousands of MACRO calls, took only 14 seconds to assemble and link on a 40MHz 68030 Amiga 1200! Tandem assembles and links its own source code (a file of over 350,000 bytes) in less than 4 seconds!

7. Special features.

Tandem has all the special features of other assemblers:

conditional assembly, include files, macros, local labels, and elaborate expression syntax and operand types.

8. Debugging.

Tandem is a symbolic debugger, as well as an editor-assembler. That is, it allows you to refer to labels and line numbers in source code, when debugging machine code. Breakpointing and single stepping can be done. Tandem is much easier to use than ROMWack.

A (unique?) feature of Tandem is that you can do mc instructions in "immediate" mode, just like doing immediate mode in AmigaBASIC. This is a very convenient and powerful debugging tool.

Tandem's debugger is somewhat limited in that your program shares Tandem's process structure, and it does not use interrupts(!), and does not use supervisor mode. But it is so easy and convenient, it has the capacity to transform assembly language programming.

9. Tandem's User Interface.

Tandem's user interface is designed for the minimum of keystrokes, and to give helpful messages if you make an error. It makes full use of on-screen editing, mouse clicks, gadgets, etc.

10. Pre-Assembled Amiga OS3.1 INCLUDE (Header) Files.

Tandem has a special feature: if you include the line

Include 'IncAll.i'

It will contain the data of all Amiga INCLUDE and FD files, which means that you won't need to declare XREF's and EQU's for all the Amiga constants - they are all built-in, as if you had included several thousand XREF's, EQU's, and MACRO's. But they are instantly incorporated into your assembly!

Compatibility with other Assemblers

Here is the only known incompatibility with Metacomco's traditional Amiga assembler, or Motorola's 68030/68881 standard:

Since Tandem assembles directly into memory, it cannot assemble programs designed to be scatter-loaded, or overlaid. This means that:

SECTION

statements are unsupported. Most assemblers (such as Devpac for example) allow the user to mix up data and machine code, just like Tandem does.

The next release of Tandem will fix this shortcoming. You will be able to load object modules into segments, and Tandem will link them together so you can save them all together, or debug them as a group.

So, in other words, I am working on it.

Apart from the above, Tandem is fully compatible with all other assemblers. Tandem supports most of the sundry extensions to assembler directives that other assemblers have made.

You will always be able to re-use source code written for other assemblers without modification, provded it is not divided into SECTIONs.

1.15 basic

Basic Concepts

Tandem is an integrated Editor/Assembler/Debugger. Read each button in this section, for a conceptual introduction to the way Tandem works.

Source Code (sc) and Machine Code (mc)

Tandem's Screen

Switching Between Screens

Tandem's Working Environment

Using the Keyboard

Programming Considerations
 <-- Important!!

1.16 scmc

Source Code (sc) and Machine Code (mc)

Tandem keeps "source code" in ascii, in a form which can be edited by any text editor such as the Amiga ED text editor. In this manual, I will from now on refer to source code as "sc".

sc (i.e. source code, remember) can be loaded and saved, and Tandem has its own built-in text editor called "Multiline" for editing sc in situ.

If you request Tandem to assemble, then providing there are no errors, not only sc but mc (machine code) will exist in memory. In this manual, I will from now on refer to machine code as "mc". You can save mc to disk just as you can save sc. The mc file you save from Tandem can then be run from the Shell, by typing its name as a command, or from the Workbench by clicking a tool icon with the same filename and .info appended.

If after assembling, you then go back and change the sc, then as soon as you change the sc, the mc will disappear from Tandem's memory until you re-assemble with the new sc. You can also save the mc as object code if you want to. (You would then need a linker to combine 1 or more object code files into an mc file; you usually don't need object code files).

1.17 screen

Tandem's Screen Usage

By default, Tandem opens its windows on the workbench. However, Tandem can also use its own screen.

Tandem has three windows, which are always open, as follows:

1. Tandem's
Main Window
 allows you to assemble and do
debugging operations.
2. Tandem's
Edit Window
 allows you to edit the sc.
3. Tandem's
Jotting Window
 is available as a scratchpad.

The windows are adjustable in size - for example, you might sometimes find it convenient to have the Edit and Jottings windows covering half each of the screen, so you can see both at once, and go from one to the other by clicking. The main window hoever has a fixed width of 640 pixels, whereas the other two can vary their width. All can vry their height.

There are menus for each of the windows, and you can switch between windows by depth arranging and clicking, or by menu selection, and in the case of the Main window also by clicking the relevent button.

The menus for all the windows contain right-Amiga hotkeys (in the usual fashion) for frequently used menu selections, and in addition the Main window has visible buttons for most of the Menu selections. I hope you find Tandem's user interface convenient and intuitive.

1.18 swit

Switching Between Screens

If Tandem has its own screen...
and you want to temporarily exit from Tandem's screen, just press <left Amiga> with <m>. This hotkey is built into the Amiga's operating system. So, for example, you may want to go back to the workbench from Tandem, when you would press <left Amiga> with <m>. You can then get back to Tandem by again pressing <left Amiga> with <m> to bring Tandem's sreen to the front again.

Alternatively, you could keep the workbench in front of Tandem, but dragged down to the bottom of the display clip, and just pull it up when required.

On its Main window, Tandem displays preliminary information, if no machine code exists. But if you do an assembly (even if it contains errors), then an assembly list will appear on the Main window.

1.19 env

Tandem's Working Environment

Tandem presents a working environment somewhat similar to that of BASIC, but complicated by the existence of both source code and machine code. The environment has three windows:

1. Tandem's
   Main Window
   allows you to assemble and do
   debugging operations.
2. Tandem's
   Edit Window
   allows you to edit the sc.
3. Tandem's
   Jotting Window
   is available as a scratchpad.

You switch between these windows as required; you can get to the Edit or Jotting window from the Main window by menu selection or by clicking the relevent button. You can return to the Main window from the Edit or Jotting windows by menu selection. And you can also switch from one window to another by clicking any of the windows to activate it, or pressing <Esc>.

In a typical session, you might start up, and go to the Edit window to load an sc file. You might also go to the Jotting window to load a doc file for the sc. You might then go back to the Edit window, and make some mods to the sc. Then, back to the Main window and assemble. Then, do some stepping and running of the mc, switching back to the Edit window to make corrections, and back to the Main window for re-assembly. When the mods are working, you might do a final assembly, and save the mc. Finally, you exit from Tandem.

1.20 keys

Using the Keyboard

Tandem's main window has many buttons, which you can click. Or else, you can select the same things by using the right mouse button menu. The menu items have keyboard hotkeys shown (in the usual way) for bypassing the buttons (or menu) for frequently used things, with RightAmiga-Key combinations.

On the Main window, if an assembly list is shown, then if you click the left-hand 4 characters on any mc line, you will go to the Edit window, with the cursor placed on that line. Handy for error correction.

You can at any time press the Help key, which will cause Tandem to display some on-line help. There is also a Guide button to consult this Amiga.guide for more detailed information.

When you select any item as above, Tandem will perhaps display a requester, of which Tandem has various types. If a requester is showing, everything else stops until you respond to the requester. In all requesters, you can press Return instead of clicking OK, or press Esc instead of clicking Cancel. With multiple choice-type menus, there is a series of boxes to click a choice, and each box has in it a Function key which you can press instead of clicking.

Hopefully, the Tandem user interface is friendly and intuitive, in accord more-or-less with the Amiga style manual. The mechanics of using the various requesters should hopefully be obvious.

Special Keyboard Combinations

Wherever you are typing characters, you can press:

shift/backspace	to delete all characters on a line, back from the cursor
shift/del	to delete all characters on a line, forward from the cursor
Ctrl/X	to erase all characters on a line, without deleting it.
Help	for online Help

Press <Help> while editing text for further particulars.

1.21 prog

Programming Considerations

Programming under Tandem is the same as for any other editor-assembler. The following data will help you with programming in such a way as to make debugging easy.

General notes for Programmers

Running Tandem from the Workbench

Writing a Front End

Finding the PROGDIR:

SECTION

INCLUDE and IncAll.i

Relative Branching

1.22 gen

General Notes for Programmers

Here are notes about the way your programs need to be written, to get the most out of Tandem. There are no particular constraints about assembling &c with Tandem - Tandem can assemble any program at all, and save it as an object file, or if it's free of unresolvable XREF's and XDEF's, as a machine code (executable) file.

But if you wish to debug with Tandem, there are a number of considerations, viz.:

1. As it is supplied, Tandem runs under the CLI (with an C:ICONX icon). Your program must be capable of executing under the CLI, if Tandem is running under the CLI. See
   Writing a Front End
   . If Tandem
   is running under the workbench, and your program might try to receive the workbench startup message, see
   Workbench
   .

2. Tandem cannot single step privileged instructions, or operate in supervisor mode. You cannot set breakpoints where your program might be

in supervisor mode, or where it has IntuitionBase or DosBase locked.

3. If your program CD's to its PROGDIR:, as many programs do, then see

Finding the PROGDIR:
(important!)

4. Tandem cannot assemble or debug programs designed to be scatter-loaded.
It always assembles in one big hunk. See
SECTION
.
(The next release of Tandem will fix this).

5. It is certainly not compulsory to use Tandem's INCLUDE 'IncAll.i'
facility: see
INCLUDE
. But, it is so good, you will find
that you always do use it. This will then mean that all labels
defined in the Amiga OS3.1 .i header files are already defined before
your program begins, so you cannot re-define them.

6. For the confusion caused by Motorola over .L extensions to Bcc and BSR
opcodes, see
Relative Branching
.

1.23 fend

Writing a Front End

n.b. I have written a suggested front end for your programs, which I call
Front.i, and which you will find in Tandem/Includes. You can use this, or
study it and if you wish modify and rename it. In any case, when you have
a well-debugged and nicely customised frontend, it is logical to put it in
Tandem/Includes as a .i file. Be sure to also read:

CD to the PROGDIR:

Starting Tandem from the Workbench
If you are writing a program to operate only from the CLI there is ↩
no
particular problem about starting up your program. But if your program is to
operate from the workbench, it must have a "front end" where you receive a
message from the workbench, and then a close down routine where the message
is replied to.

So, in particular, your program must start up like this:

```
    * start things up

     INCLUDE 'IncAll.i'      ;define all OS3.1 data & MACRO's pre-assembled

     BRA coldstart           ;begin execution at coldstart
```

```
    initd0: DS.L 1           ;store initial D0 here   } Parameter len & addr
    inita0: DS.L 1           ;store initial A0 here   } if under CLI
    workbench: DS.L 1        ;workbench startup message, or 0 if from CLI

    coldstart:
     MOVE.L D0,initd0        ;save initial D0    }Parameters len,address
     MOVE.L A0,inita0        ;save initial A0    }meaningful if under CLI
     MOVE.L _AbsExecBase,A6  ;point to exec.library base
     SUB.L A1,A1             ;A1=0 to find current task
     JSR _LVOFindTask(a6)    ;find own task structure
     MOVE.L D0,A2            ;which put in A2
     CLR.L workbench         ;workbench=0 if started from CLI
     TST.L pr_CLI(A2)        ;started form CLI?
     BNE.S fromcli           ;yes, go
     LEA pr_MsgPort(A2),A0   ;no, point to own message port
     JSR _LVOWaitPort(a6)    ;wait for startup message
     LEA pr_MsgPort(A2),A0   ;it's ready, so again point to message port
     JSR _LVOGetMsg(a6)      ;get the startup message
     MOVE.L D0,workbench     ;workbench<>0 if I was started from Workbench
    fromcli:                 ;now ready to open libraries, &c
```

Then, your program does its thing, & finishes like this:

```
     TST.L workbench         ;did I start from workbench?
     BEQ.S return            ;no, go
     MOVE.L _AbsExecBase,A6  ;get sysbase
     JSR _LVOForbid(A6)      ;forbid multitasking until RTS
     MOVE.L workbench,A1     ;retrieve the workbench startup message
     JSR _LVOReplyMsg(A6)    ;reply to it
    return:                  ;all done! so...
     MOVEQ #0,D0             ;signal ok
     RTS                     ;exit back to CLI/workbench
```

Note the use of INCLUDE 'IncAll.i' above - see
INCLUDE
- so
giving you access to all _LVO's and OS3.1 header .i files, without any
further INCLUDEs or XREFs.

In the
Tandem.i
section of this Manual, you will see data
about a more comprehensive front end which you can use, called Front.i

1.24 rel

Relative Extensions

Motorola has caused a confusion about the extensions of Bcc and BSR
(relative branching) opcodes, as follows:

(a) unfortunately, in 68000 assembly language, byte and word relative
addresses were given the extensions:

.S for short (but it should have been .B)

```
    .L  for long     (but it should have been .W)

(b) then, in 68020+ assembly language, when longword relative branching
    became possible, the extensions became:

    .B  for byte     (same as .S)
    .W  for word     (same as .L)
    .L  for longword (new to 68020+)

This caused massive confusion to upward compatibility with 68000 programs,
since all their Bcc.L and BSR.L opcodes were not meant to be
longword.

Each assembler deals with this in different ways. Here is what Tandem does:

1. in the case of backward branches, like most assemblers, Tandem ignores
   the extension, and simply optimises it. However, if you have set "Yes"
   for the Preference under 4. below, Tandem will not use .L, but will
   optimise for .B or .W, and .L would be out of range (if >32766 bytes away)

2. in the case of forward branches, if they have no extension, Tandem
   assembles them as word.

2. in the case of forward branches, tandem assembles .S and .B as byte.

3. in the case of forward branches, tandem assembles .W as word.

4. in the case of .L, Tandem assembles them as .W  (!!) - however -
   if you select
                Prefs
                 on the Main window, you can switch
   the following preference:

   Change .L forward references to .W

   between "Yes" and "No".

Also, you can use the following facility: after a successful assembly, you
can select
                Rel exts
                 from the Main window. You can then
select one of:

   Next Bcc/BSR .L in assembly list     (hotkey Ctrl/L)

         When you can scan through these, changing any of them to .W as
         required. If they are to be 68000 compatible, that will be
         all of them.

   Next Bcc/BSR .S in assembly list     (hotkey Ctrl/S)

         When you can scan through these, changing all of them to .B in
         accord with reformed 68020+ assembly language dialect.

Thus, you can update all your sc files, and then having done that, save the
preference:
```

```
   Change .L forward references to .W     No
```

(Updating should be done in the context of an assembled program, to pick up extensions inserted by default, or synthesised from MACRO parameters).

Then, you'll need to get into the habit of using .B and .W instead of .S and .L when writing new programs. Hard!

I hope the above seems to you to be a reasonable solution to the problem.

1.25 cdpd

CD'ing to the PROGDIR:

One of the big improvements in Amiga OS2.04 is that it became easy to find a prgram's PROGDIR:. Thus, a program could be stored in a directory, with all its supporting files, and as long as the dir was kept together, it could be stored anywhere at all, and all the supporting files easily found. This is one of the reasons OS1.3 became so quickly obsolete.

Most programs do this by CD'ing to their PROGDIR:. A program must then, before it closes down, CD back to the original CD it had when it began operation, or it will crash.

Here are the relevent methods:

```
   MOVE.L dosbase,A6       ;base of dos.library to A6
   JSR _LVOGetProgDir(A6)  ;get PROGDIR:
   MOVE.L D0,D1            ;into D1
   JSR _LVOCurrentDir(A6)  ;make it the CD
   MOVE.L D0,oldcd         ;cache old CD
```

All loads and saves can then be relative to the program's own CD, where its supporting files are saved. Also, the Amiga's path always includes the CD, which means that commands, libraries and the like used by your application can be in the program's dir, and need not be put in C: or LIBS:.

Finally, when it closes down, your program must do this:

```
   MOVE.L dosbase,A6       ;base of dos.library to A6
   MOVE.L oldcd,D1         ;retrieve aboriginal CD
   JSR _LVOCurrentDir(A6)  ;restore CD as when program started
```

Now this is all very nifty and simple. But, a problem arises when your program is being debugged by Tandem. In that case, since your program runs under Tandem, _LVOGetProgDir will return Tandem's PROGDIR:, not your final PROGDIR:. So, here's what you have to do:

1. On Tandem's main window, you will see a button:

   dbug CD

   Click the button, and a requester comes up, asking you to specify the directory when debugging. Specify the full path of where your project's supporting files are placed.

2. Then, when Tandom is debugging, when you either single step or run,
   before Tandem single steps or runs, it will CD to the dbug CD you
   specified above. And, when it returns, Tandem will CD back to its own
   PROGDIR:

This solves the problem nicely.

Am I being Debugged?

Does your program want to know if it is being debugged by Tandem? I have
built a little kludge into Tandem, so your program can tell. Here's how
it works.

If you make the very first mc instruction of your program to be:

```
    TST.L  (A7)
```

then of course the CCR will be NE, since (A7) can never be zero. But, what
Tandem does is this. If it finds TST.L (A7) at the start, it jumps the
debugging PC past it, and sets EQ. Since this is impossible, it will tell
your program that it is being debugged by Tandem, rather than running
normally. Here is some sample code to show what I mean:

Determine if your program is being debugged by Tandem

```
ColdStart:
 TST.L (A7)                ;set NE if running normally
 SEQ D7                    ;this always sets 0, but if debugging will be -1
 EXT.W D7                  ;make word length
 MOVE.W D7,debug           ;will be 0 normally, or -1 if debugging

 ......

debug:     DS.W  1         ;gets set to -1 if debugging, else 0
 ......
```

1.26 edwin

The Edit Window

The Edit window is used for loading, saving and editing 680x0 assembly
language. To switch between windows:

- from the Main window, click the Edit sc button, or select
  Edit sc from the Menu (hotkey RtAmiga/E); alternatively, click
  the Edit window to activate it.

- from the Edit window, click the close gadget, or press Esc, or select
  Stop editing from the Menu (hotkey RtAmiga/Q); alternatively,
  click either of the other windows to activate it.

You use the Edit window to type in text, and edit it, just like any other text editor (like ED for example). I call the type of text editor built into the edit & jotting windows the "Multiline" text editor. When debugging is in progress, and you activate the Edit window, Tandem will ask you if you wish to Lock the text. If your debugging has allocated any resources, you should answer yes, or else you might inadvertantly change the sc, which will cause the mc to vanish, leaving the resources locked up.

The buttons below explain the various menu items:

The Edit Menu
Important...

The Go To Menu
When the Edit window is active,

The Cut & Paste Menu
you can select "View AmigaGuide"

The Page Format Menu
from the "Project" menu to see a

The Line Format Menu
guide with more details than here.

1.27 edmen

The "Edit" Menu

The "Edit" menu contains the following selections:

1. Load	for loading a source code file
2. Save	for saving, to the same filename last used
3. Save as	for saving, after a filename requester
4. Print	for printing some or all of the sc
5. About	for getting the version number of the text editor
6. GUI preferences	for setting prefernces about pens, requesters, &c
7. View AmigaGuide	for viewing an AmigaGuide about the text editor, in the file Tandem/Multiline.guide Multiline.guide contains more info than here, but the extra info is not relevent to the way Multiline is used in Tandem.
8. Lock/Unlock Text	choose lock to allow you to peruse text without being able to edit it, or unlock to enable editing. If you have long lines, lock activates the horizontal scroller. (If you choose "Lock sc" when you activate the Edit window, "Unlock text" will be disabled).
9. Stop editing	for going back to the main window

All of the above operate in accordance with normal Amiga usage. In fact, you could just about get away with not reading this manual for the Edit and Jotting windows. Are you there?

I suggest that you:

1. Save all your sc files in the "Projects/Asmfiles" sub-directory of the "Tandem" directory. And make subdirs in "Projects" for other supporting files for each of your projects. This will include an AmigaGuide for users of each of your your projects.

2. Use the suffix .asm for all sc files. (Some programmers use other suffixes such as .r or .a)

3. Create subdirs in "Jottings" for the development docs for each of your projects.

I write the AmigaGuide for users of my projects before I begin work on them - I always imagine I'm writing for the general public, even if it is for my private use. Writing the user AmigaGuide before I start work (as I am doing now with Tandem) helps me to be clear on what exactly I am doing. The Most important part of designing any program, is designing its user interface!

Tandem loads and saves text in the form of plaintext files: i.e. they are all printable characters, with a line feed ($0A) at the end of each line. The only exception is that Tandem will remove tabs from lines if it finds them, just like ED does, but does not save lines with tabs set. No effort is made to compress the saved file in any way.

No line of sc can be more than 254 characters in length. All lines longer than that will be split. Lines too wide to fit on the window will scroll across the window with the cursor. (the horizontal scroller only works if you "lock" the text ("Lock/Unlock Text" in the "Project" menu).

Unlike ED, which refuses to load bad characters, Tandem will load anything whatsoever, and remove all unprintable characters, and split any too-long lines. Since any uncompressed file of almost any format will contain text as ASCII, Tandem will more-or-less load the text in any sort of format, and it is then up to you to remove any garbage loaded along with it. Thus, you would actually get something in memory if you loaded, say, a sound sample, but of course it you tried to assemble it it wouldn't make a lot of sense. This feature allows you to perhaps work on recovering corrupt plaintext files. Tandem doesn't support datatypes for loading text.

When you already have sc in memory, and you select "Load", Tandem will entirely overlay existing text with whatever is loaded. So if you want to insert (import) text from another file somewhere, but keep your existing text, then select "Insert file" from the "Cut & Paste" menu.

1.28 goto

The "Go To" Menu

The "go to" menu is used to jump about in the source code. The practical limit for source code editing is about 1 meg in size, if you have an Amiga 1200 with an accelerator; beyond that, response time slows down, and you'd need a more sophisticated text editor.

The "go to" menu has the following selections:

1. First line	jump cursor to first line of sc
2. Last line	jump cursor to last line of sc
3. Seek Forward	seek forward for a string of characters
4. Seek back	seek backward for a string of characters
5. Seek at line start	seek for a string of characters, at the start of lines only (i.e. labels)
6. Forward a window-full	jump cursor forward a window-full
7. Back a window-full	jump cursor backward a window-full
8. Go to line number	jump to a specified line by number
9. Info about text	see info about mem used, lines &c.

If you specify a null string for seek forward, seek back, or seek at left, Tandem will use the previous input. Thus you can repeatedly seek the same string by sending subsequent null inputs for the string sought.

You will soon learn the hotkey menu bypasses for the above menu selections, because you will use them all a lot.

Selecting an mc Line for Editing

If you are on the Main window, and an assembly list is showing, you can click within the left four characters of any line and (provided it is not in an include file) Tandem will put you in the Edit window, with the cursor placed on the sc of the line you clicked. Useful!

1.29 cpast

The "Cut & Paste" Menu

The "cut and paste" menu options are used for splitting and re-arranging sc files, &c. the options are as follows:

1. Insert a line

   You can of course insert a line in sc by pressing the Return key in the normal way. You can also select Insert a line in the "cut and paste" menu, or of course by using its hotkey (i.e. RtAmiga/I).

2. Delete a line

   You can delete a line in sc by selecting Delete a line in the "cut and paste" menu, or of course by using its hotkey (i.e. RtAmiga/D).

3. Mark Start of Range

   In order to do things with a range of lines all at once you need first to mark the range. To do that you should select Mark Start of Range with the cursor on the first line of the range. Then, you would put the cursor on the last line of the range, and select:

   Mark End of Range

The range can be only 1 line, with start and end the same line, but of course the end cannot be before the start. If you do anything to change the number of lines in memory (other than Insert Range), the range will become unmarked. But if you merely change the contents of particular lines, without changing the number of lines, the range will stay marked.

4. Delete Range

Once a range is marked, you can select Delete Range to make it disappear from memory. (Careful!).

5. Insert Range before Current Line

Once a range is marked, you can place the cursor on some line, and select Insert Range which will duplicate the range before the line containing the cursor. A range cannot be inserted within its own compass. More often than not, you will immediately select Delete Range after Insert Range, to remove it from its old location.

6. Save Range

Once a range is marked, you can save the range to a diskfile by selecting Save Range.

7. Insert File

Inserts (imports) a file into the existing memory, before the line containing the cursor.

8. Rewrap

If you have a series of lines, you can cause them to be re-wrapped into a compact paragraph by selecting Rewrap. You can choose whether to re-wrap from the cursor line to the end of its paragraph, or the entire of memory, or a marked range (if a range is marked).

Presumably you would never use this for source code, but you might for jottings.

For more particulars about rewrapping, see the Multiline.guide

9. Change maximum line length

You can set this to values from 20 to 254, but in general I recommend 76.

10. Spell Check range

This is always disabled in Tandem.

11. Erase ALL Lines (careful!)

Select Erase ALL Lines to NEW the sc. Use with care!

1.30 style

```
The "Page Format" Menu

These menu items are all disabled for Tandem.
```

1.31 lform

```
The "Line Format" Menu

Most of these are disabled, except for:

Erase (x out)  This item (keyboard Ctrl X) erases all characters on a line
               i.e. it "x'es out" a line
Undo           This item (keyboard shift Ctrl U) undoes the effect of the
               previous keystroke (even if it was Undo or Restore)
Restore        This item (keyboard shift Ctrl R) restores the line to how
               it was when the cursor first landed on it.
Space fill     This item fills as many spaces as will fit behind the
               character under the cursor, thus pushing all from the cursor
               onwards to the right.
```

1.32 jtwin

```
                The Jotting Window

The Jotting window is intended entirely for your use as a scratchpad. You
load from and save to it, and edit in it, just as you do in the

                Edit Window
                 which you should refer to for instructions about
editing, should you need them. Tandem does not assemble what is in the
Jotting window, it is just used by you (if you want to) as a scratchpad, or
for simultaneous viewing of docs with editing of sc.

I would suggest that you use the "Jottings" sub-directory in the Tandem
directory, for your jottings; you can of course divide it up hierarchically
into categories.
```

1.33 mnwin

```
                The Main (Debugging) Window

When you first start up Tandem, you will see the Main window, covering
Tandem's screen. The buttons below correspond to the buttons and/or menu
selections on Tandem's Main window.


                Edit sc
```

Step mc

PC line

Jotter

Run mc

Memory

Mem buff

Save mc

Break pt

Assemble

Llist mc

Registrs

View sc

Calc

Save obj

Guide

View sym

Sv custm

Halves

Rel exts

Go to

dbug CD

Prefs

Immed

Clk left

Modules
The Main window initially contains preliminary information.

If you conduct an assembly, the preliminary inforation is replaced by an assembly list on the window. If the assembly is error free, then mc will exist, and all the buttons of the window will be available. If there is no mc, then some if the buttons are active, and some not. If an assembly fails, an error list will appear, and you can go to the corresponding line in sc

to the errors, by clicking their left-most 4 characters.

Here is a summary of the functions of the above buttons and options:

```
 1. Edit sc  Peruse or edit the sc
 2. Jotter   Peruse or edit the jottings
 3. Mem buff Peruse or change the program memory buff preferences
 4. Assemble Assemble the sc into mc
 5. View sc  Peruse sc or include files
 6. Guide    Peruse Tandem.guide (what you ar reading)
 7. Halves   Arrange Main, Edit and Jottings window frames
 8. Prefs    Peruse or change program prefs (see also Mem buff)
 9. Step mc  Single step the mc when bebugging
10. Run mc   Run the mc when debugging (& breakpoints set)
11. Save mc  Save the mc just assembled
12. Llist mc Send an assembly list to the printer
13. Calc     See the value of a symbolic expression
14. View sym View the values of symbols
15. Rel exts Lists of relative addr exts, >68000 mc, privileged &c
16. Immed    Do an "immediate mode" assembler instruction
17. PC line  Jump the debugging PC to a particular mc line
18. Memory   View a block of memory
19. Break Pt Set or clear a breakpoint
20. Registrs Change the contents of registers (except A7)
21. Save obj Save object code (rarely used)
22. Sv custm Save custom symbols (rarely used)
23. Go to    View specified places in the assembly list
24. Clk left Go to edit sc, with the cursor on the line clicked
25. Dbug CD  Set a CD for your program during debugging
26. Module   Divide programs into modules (not operative yet)
```

All of the above make for a convenient, flexible and responsive working environment. You will find especially that debugging is much, much easier, and that you don’t need to print out assembly lists and symbol tables.

1.34 pref

```
                Memory Buffers

You can do two types of things by selecting Mem buffs on Tandem’s
Main window:


                Flush the INCLUDES buffer

                Resize the Memory buffers
                See also


                Stack Usage, ICONX, &c
                If you click "Save", [or alternatively within the
                Prefs
                option], Tandem in either case saves the current memory sizes or  ↩
                   sundry
preferences in a directory called:
```

ENVARC:Tandem/Tandem

which Tandem will find and use (if it exists) to set initial buffer sizes and sundry options, each time Tandem runs. This is just like the Preferences window in the Workbench.

When Tandem warms up, it tries to use the currently operative memory buffer sizes as they are built-in to Tandem, and as modified in ENVARC:Tandem/Tandem if such exists. If the memory buffers will all fit in the available memory, well and good (they all go in public memory).

If not, Tandem downsizes the buffers progressively, until a successful fit is made. You can at any time re-size the memory buffers by selecting "Mem buffs", and adjusting the sizes, and then selecting either:

Use To re-size the buffers only until the Amiga is switched off (saves to ENV:Tandem/Tandem)

Save To re-size the buffers, and also save the new sizes in ENVARC:Tandem/Tandem

Cancel To leave all the buffers alone

Hopefully, few users will ever need to resize the memory buffers.

If you select Use or Save, Tandem wil try to allocate the sizes you specify, and will if necessary keep downsizing them until they fit.

Incidentally, if you select Use or Save with everything the same, Tandem will leave the buffers alone, but will still save to ENVARC:Tandem/Tandem if you clicked "Save".

If you are fair dinkum about doing program development, then you really need 2 megs of chip ram, and say 8 megs of fast ram, such as you get in an Amiga 1200 with an accelerator card added. But, in principle, Tandem will work with an Amiga 500 with 1meg total (the editor is a bit slow).

Note: if you have done all sorts of things which have made the memory become fragmented, then there is a slight chance that Tandem will not succeed in resizing the buffers after Use/Save. Tandem will then pause for acknowledge and close down. This is very unlikely to ever happen.

If you resize any buffer from "Source code" up, Tandem will zap the source code when you select Use or Save. So be careful to save sc before pressing Use or Save!

If you resize any of the buffers other than "Debug Stack", Tandem will zap your jottings when you select Use or Save. So be careful to save jottings before pressing Use or Save!

When you see the table of buffer sizes, Tandem shows you not only the existing sizes, but the amount of actual usage as at the most recent assembly. This allows you to make sensible decisions based on the usage of this particular project as the sc currently stands.

1.35 flush

```
                Flushing the INCLUDE Buffer

Select Mem buffs on the Main window, and the window will contain
(among other things) the Flush Includes button.

For flushing the Include buffer, see also the
                INCLUDE
                 pseudo
op. Tandem always keeps IncAll.i in the buffer, and any other .i files any
of your programs ever assemble. There they stay until Tandem closes down.

But if you want to remove all of the .i files from the Includes Buffer, then
click the Flush Includes. This removes all .i files, except
IncAll.i

You can see which .i files are in the buffer with
                View sc
                from the Main window.
```

1.36 resiz

```
                Setting Memory Buffer Sizes

Tandem is a memory hungry program, since its assembler is designed for speed
at the cost of memory. Here is an explanation of Tandem's memory buffers:

Tandem rounds each buffer size to a multiple of 16, and gives it a minimum
of $2710 bytes (about 10,000). Note that Tandem always loads incall.consts,
ssxref.consts and opcode.consts into memory before it creates the buffers
below, and you'll see what I mean by memory hungry.

1. Labels pointers       (typical value $3000)
   Labels ASCII          (typical value $10000)

   These are used to store labels during an assembly. Each label assembled
   requires 4 bytes in Labels pointers, and (its length in ASCII) + 7 or 8
   bytes in Labels ASCII.

2. Includes              (typical value $80000)

   Used for holding .i INCLUDE files. Remember, you need never use the OS3.1
   INCLUDEs. So, you may wish to make this smaller if you don't wish to use
   many includes. (See
                Includes
                ). Tandem always has IncAll.i
   in this buffer, even if you don't use it.

3. Source code           (typical value $100000)

   Used to hold your sc

4. Object code           (typical value $20000)
```

Used to hold your mc

5. Assembly Lines (typical value $80000)

This holds 16 bytes for every line of the assembly list, including comment lines, INCLUDEs and MACRO expansions. Used for cross-referencing during debugging.

6. Relocations (typical value $3000)

This holds 4 bytes for every relative address assembled in any expression.

7. Xdefs & Xrefs (typical value $2710)

This holds any XDEFs and "private"
XREF
s assembled. Each
XDEF takes 1 entry, and each XREF takes an entry every time it is referenced. Most users will never use this buffer. It is exceedingly unlikely that any program would ever overflow the minimum size.

8. Pass 2 push (typical value $30000)

This is to make the 2nd pass fast. It holds the ASCII of every expression containing a forward ref, plus about 18 bytes extra per expression.

You can make your assemblies faster (and this buffer smaller) by putting a BRA to the start of your mc at the start ot your program, and then putting all your DS's and DC's, and finally all the actual ops.

9. Jottings (typical value $10000)

Holds the ASCII of your jottings. Some users may want to make this larger for large doc files.

10. Debug stack (typical value $2710)

Your mc's stack while debugging. The RKM's say to allow 4096 bytes for Amiga _LVO calls. See also
Stack Usage
.

If you use front.i/tandem.library, then 4906 will just suffice, provided you make minimal use of your stack - see
Stack Usage
for details.

When you look at this table, you will see the current sizes, and the amount of each buffer currently in use. You can edit the buffer sizes. You can also use the following buttons:

1. Proportional to sc - original

This takes the value you have typed in for the "source code" buffer, and adjusts the following buffers:

```
   Labels pointers
   Labels ASCII
   Object code
   Assembly lines
   Relocations
   Pass 2 Push

   to be in the same proportion to it as the "typical value"s above.

2. Proportional to sc - existing

   Same as "Proportional - original", but adjusts in proportion to the
   current buffer sizes.

Use of one of these buttons enables you to adjust the 6 buffers quickly in
sensible proportion to the source code buffer.

When Tandem warms itself up, it will downsize the buffers from the
preference sizes if there doesn't seem to be a lot of spare memory.
```

1.37 stak

```
Your Program: Stack Usage and ICONX

Making an ICONX Icon for your Program

I find ICONX to be a good way to start my programs. Here is how to do it.
Let's suppose you have a program called "Fred".

1. on the CLI,   Makedir  WORK:Fred

2. Use Amiga's "IconEdit" (in workbench tools) to make an icon for the
   Fred directory, and save it as a "Drawer" icon (see the Types menu
   in IconEdit) as   WORK:Fred.info

3. Write an AmigaGuide for your program before you write your
   program. Save it as  WORK:Fred/Fred.guide

4. Make an icon for your guide, and make it a "Project" icon. Save it as
   WORK:Fred/Fred.guide.info

5. On the workbench, click the Fred draw open, and click the Fred.guide
   icon once so it is highlighted. Then, choose "Information"
   in the workbench "Icons" menu. You will see a window come up about
   your icon. In its "default tool" box put "Multiview" and click Save.

6. Make docs for your program using  Tandem/Jotter, and save the docs as
   WORK:Tandem/Jottings/Fred/Fred.docs

7. Write your sc for fred, and save it as
   WORK:Tandem/Projects/Fred/Fred.asm

8. If there are supporting files for Fred, then put them in the Fred
   directory. Whenever you are debugging Fred, click the "Dbug CD" button
```

on Tandem's main window & enter WORK:Fred

9. When you assemble Fred and it can run, save it as
   WORK:Fred/Fred

10. use Jotter to make a with file just 1 line

    Fred

    in it, and save it as   WORK:Fred/Fred.exe

11. Now use IconEdit to make an icon for Fred, and save it with type
    "Project" as   WORK:Fred/Fred.exe.info

12. In the workbench, bring up the "Information" window for
    WORK:Fred/Fred.exe

    In the "Stack" box type 4096

    In the "Default Tool" box type   "C:ICONX"

    In the "Tool Types" box type (using the "New" button):

        WINDOW="CON:0/50//130/Monitor/CLOSE
        STACK=4096

Now, your development environment for Fred is all set up.

Stack Usage

All the Amiga library _LVO's are guaranteed to use less than 4096 bytes of stack when you call them. If you use a ICONX project icon (which I always do for my programs), then it has a default stack size of 4096 bytes. This will be enough to make _LVO calls, and for your subroutines to push all the registers down to say 10 deep, as well as use about 2048 bytes total at any one time for other things.

But, to be sure, do this:

1. assemble your program - then look in "Mem Buffs" under debug stack. You will see there that your program uses a small about of memory, being the registers &c which Tandem's debugger uses.

2. now, run your program, getting it to do whatever you fancy will use the most stack - in general, try to put it through its paces. In particular make sure it calls all possible _LVO's within it.

3. Finally, look again at the debug stack usage. You will see the total amount of stack your program used. (Tandem works this out by filling up your stack memory after it assembles with $BADFACED, and then after it runs, it looks to see how many consecutive $BADFACED's it finds from the bottom).

Make sure this is well under 4096, e.g. make sure it does not exceed $0F00. If it does, then you must put the following on your ICONX icon's

```
information window:

       In the "Stack" box           8192           (or whatever)
       In the "Tool types" box      STACK=8192


Front.i / tandem.library stack usage

Front.i adds 1024 bytes straight away to your stack usage. If you call
tandem.library routines, then some of them (especially help with a Guide
button, and TLmultiline) use so must stack, it is dangerous to ahve only
2048 bytes in your stack, and you should make it 8192 to be sure. Stack
overflow can be a puzzling bug to track down.
```

1.38 sundry

```
                Sundry Preferences

If you click "Save" within
                Mem buff
                , or the Prefs
option, Tandem in either case saves the current memory sizes or sundry
preferences in a directory called:

   ENVARC:Tandem/Tandem

which Tandem will find and use (if it exists) to set initial buffer sizes
and sundry options, each time Tandem runs. This works like the Preferences
window in the Workbench.

(Note: if you choose save "GUI Preferences" in the Edit or Jotting windows'
Project menus, prefs will also be saved in ENVARC:Tandem/GUI).

To set Sundry Preferences, select "Prefs" from the Main window.
You will then see a table of buttons, with the preferences as currently
operative. You can then re-set the preferences as you desire, and finally
click:

  Use     To use the preferences as you have re-set them.

  Save    To use the prefs as re-set, and also save to ENVARC:

  Cancel  To leave the sundry prefs alone

These are the sundry preferences you can set:

Change .L Bcc/BSR's with Forward refs to .W     Yes/No

See
                Relative Branching
                 for the confusion caused by Motorola
over this topic. Tandem originally sets this to "Yes".

Maximum mc Bytes/Line in Assembly List
```

Tandem originally sets this to "12". This causes Tandem to put 24 spaces between the relative address and the sc in the assembly list on the window, and as it it might be
Llist
ed, to hold the mc bytes assembled
for that line. Any given actual op can theoretically take up to 20 bytes - usually much less. And DC's can take much more. DS's can too, though their contents are all zeroes. You can set the maximum mc bytes per line from 4 to 22. If you set it to more, you will see more mc (if it interests you), but comments will get chopped off. And if less, then only very long comments will be chopped off. Many users will save a value of 4 for this preference.

Leading Spaces per Line in Assembly List

This preference only applies to
Llist
ed assembly lists.
This allows you to set a left-hand margin on printed pages, but see

Llist
also about this. Tandem originally sets this to 5.

Maximum Characters per Line in Assembly List

This preference only applies to
Llist
ed assembly lists.
This allows you to set the length of lines in the assembly list. Tandem sets this at the beginning of an assembly, but any
LLEN
pseudo ops
will over-ride it.

Tandem originally sets this to 70.

After you give your input, Tandem will ask if you want a $0F character sent to the printer, if the assembly list is sent to the printer. For most but maybe not all printers, $0F causes condensed printing. If you plan to answer yes to this, you would specify say 99 for maximum characters per line.

(after printing an assembly list with $0F's, turn off the printer & turn it on again to stop it printing condensed characters).

Printable Lines per Page in Assembly List

This preference only applies to
Llist
ed assembly lists. This
allows you to set lines per page in the assembly list. Tandem sets this at the beginning of an assembly, but any
PLEN
pseudo ops will
over-ride it.

Tandem originally sets this to 72 (for A4 paper).

Suppress MACRO Expansions in Assembly List Yes/No

```
This preference only applies to
                Llist
                ed assembly lists.

Tandem originally sets this to No.

Suppress INCLUDE Expansions in Assembly List   Yes/No

This preference only applies to
                Llist
                ed assembly lists.

Tandem originally sets this to No.

Default CPU
Default 68881 Allowed

Tandem always assembles all sc without error, provided it is compatible with
68030 assembly language (with MMU and FPU also). So, in one way, this
preference has no effect.

Nevertheless, at the beginning of an assembly, Tandem begins by noting your
preference. If your program includes
                MC680x0
                 pseudo ops,
Tandem over-rides your preference.

Then, after your assembly, if you select Rel exts from the main
window, and then Next Bad 680x0 will catch any entries in your
assembly list which are incompatible with your preference.

Similarly, if you have "No" for Default 68881 Allowed, and there
are MC68881/2's in your assembly, Next Bad 680x0 will
also catch any FPU opcodes in your assembly list.

See also under
                Rel exts
                 for further details.

Screen Type

Tandem by default opens its three windows on the Workbench (default public
screen). If you decide that makes the workbench screen too crowded, you can
click "Screen Type" which allows one of three options:

  - place the windows on the workbench
  - place the windows on a non-interlaced private screen
  - place the windows on an interlaced private screen

Unlike other prefs, this pref does not take effect immediately if you choose
"Use" or "Save" - you must first close Tandem down & then reload it.

You will probably find Tandem more pleasing to use if it has its own
private screen, but if your Amiga has a graphics card, Tandem does not open
on that graphics card if it uses a private screen.
```

1.39 assem

Assembling

When you have loaded and/or edited your sc to your satisfaction, the first thing you should do is save it to disk. Then, go to the main window, and press the Assemble button (or select "Assemble" in the menu strip).

If an assembly is in mid-stream, you can abort it by moving the pointer to the top of the screen.

Tandem assembles extremely quickly. For example, Tandem assembles its own sc in about 4 seconds on a 40MHz 68030 Amiga 1200. I have made an extremely large sc file, by concatenating all the Amiga OS3.1 include files, to make a combined sc file of over 600Kbytes, with thousands of MACRO references. This monster takes only 14 seconds to assemble! Programs less than 5000 lines assemble virtually instantaneously. (A full speed 68060 should assemble Tandem's own sc in less than a second, though I haven't tested that).

Unlike other assemblers, which slow down quadratically with program size, Tandem slows down approximately linearly. that is, assembly time is roughly proportional to the number of lines of sc. Assembly is slightly slower for MACRO references, and for forward references. For the latter reason, I suggest you begin your program with a BRA to its cold start, and then put all the DC's and DS's (and MACRO's of course) before your actual ops. Most programmers do this anyway. But, even if you put all your DC's and DS's at the end, Tandem is still extremely fast.

When assembly begins, Tandem first shows:

```
    Tandem is assembling

       Pass 1...
```

Then, when pass 1 is finished, then if there are no errors,

```
       Pass 2...
```

appears. Pass2 takes much less than a second, regardless of the sc size. But if there were errors in pass 1, then pass 2 will not happen.

If errors happen in either pass 1 or pass 2, Tandem will report an error count, and wait for you to click the window, after which an error list will appear. You can click the leftmost 4 characters of any entry on the error list to go to the sc line that caused the error (if it is in a MACRO expansion, Tandem tries to put you on the line that invoked the MACRO). If you amend the sc to correct 1 error, it gets harder and harder for Tandem to keep finding your errors in sc if you click their leftmost 4 characters, so if Tandem reports it can't find your error, then re-assemble.

Note that if you correct all errors from pass 1, and re-assemble, you might get a new set of errors from pass 2.

If there are no errors, then the assembly will be successful. But, if there is no executable mc (e.g. the sc is all comments, or all DC's, &c) then the assembler will report that you can save object code, but not mc, and you

cannot do debugging.

If there are unsatisfied XREF's, (i.e. that are not XREF's to Amiga system library _LVO's), then you can save object code but not mc. Of course, if there are unsatisfied XREF's, they are most likely misspelled _LVO's. (Find any such with hotkey Ctrl/X). See
Rel exts
.

If there is executable mc, but the first thing at relative address $0000 is not an mc instruction, then once again you can save object but not mc, and you cannot do debugging.

That is to say, the Amiga operating system (at least as Tandem can save your mc) requires the entry point to your mc to be at the start. So, the first line that actually assembles anything must be executable mc, not for example a DC or DS.

Anyway, if all is well, Tandem will report

mc exists, ready to save or debug

and after you click to acknowledge, it goes back to the main window, when all debugging buttons are available (note again that if you edit the sc, the mc vanishes).

Fatal Assembly Errors

The assembly will normally continue to the end of the current pass if errors occur. But, if a fatal error occurs, the assembly will abort at that point. If there are more than 50 non-fatal errors, then that in itself causes a fatal error. Also, if one of the memory buffers overflows, or an include file cannot be loaded, or if you move the pointer to the top left of the window, a fatal error will occur.

If a memory buffer overflows, you must go to
Mem size
to
re-size the memory buffer(s).

After a successful assembly, you can click Save mc to save the mc file. This saves the mc as an executable module, which can be run from the CLI, or (if it has an appropriate frontend) from the workbench.

1.40 viewsc

View Source Code

If you select "View sc" from the main window, Tandem puts up a special requester for scanning the source code, whether it be the sc in the Edit Window, or any Include file that might be in the memory (whether or not they were included in the latest assembly).

Tandem always keeps at least IncAll.i in its includes memory buffer. If your programs INCLUDE anything, then they are loaded into memory before they are

assembled. See
Includes
for more information about includes.

1.41 guide

View Tandem Guide

If you select "Guide" from the main window, Tandem will put Tandem.guide (i.e. what you are reading) on its screen, where you may consult it, until you press the close window gadget on it. This is handy for quickly looking something up.

If you want to keep looking back and forth between Tandem and the guide, I suggest that you do the following:

1. Press LeftAmiga/M to get back to the workbench screen.
2. On there, open the Amiga.guide by clicking its icon in the Tandem drawer.
3. Then press LeftAmiga/M to switch back & forth between the workbench & Tandem screens.

Because you can do the above, I have not made Tandem.guide asynchronous on Tandem's screen, because it makes the screen too crowded.

1.42 half

Halves

In Tandem's user interface, I have included 2 editing windows:

The
Edit Window
and the

Jottings Window
.

Often, when using the Edit window, you will want to arrange the screen so that the top half is the Edit window, and the bottom half is the jottings window, and you can click back and forth, seeing both as you work, with sc in one, and reference docs in the other. Or you might even put the sc in the jottings window, to look at one part while editing another part.

In fact, you might want to see any combination of the windows at once, or even all three. To easily arrange them, press the Halves button to choose any combination of windows to see at once. You can then go from window to window by clicking them. the last option on "Halves" is to make all windows full size again.

Note: when you exit from Tandem, Tandem saves the position of your windows in prefs (i.e. ENVARC:Tandem/Wbox), and when you next run Tandem, the windows will be moved to where you had them the last time you ran Tandem.

1.43 errs

```
                Relative Extensions, & other Lists

If you select "Rel exts" from the Main window, Tandem puts up a
multiple choice requester to allow you to select from a number of choices.
Each item will allow you to scan the assembly list for the next thing in the
particular category. They are accompanied by hotkeys.

In each category, Tandem places the next instance of the category mentioned,
at the top of the Main window assembly list. Once you get to the last,
Tandem then wraps around to the beginning.

If you modify sc in any way after an assembly, Tandem will zap your mc
and/or assembly list, until you conduct another assembly.

Next Bad 680x0 in Assembly List

See Default CPU under
                Sundry Preferences
                 for
info about this.

Note that Tandem does not produce assembly errors for "Bad 680x0"s, it just
assembles everything that is legal for the 68030. So if you want say only
68000 to be accepted, you would set "Default CPU" to 68000, and then use
this option to find all lines > 68000. Its hotkey is Ctrl/N. (The mere
absence of >68000's does not guarantee that a program which runs on a 68020
will run on a 68000, since an assembler cannot detect most instances where
a .W or .L instruction might operate on an odd address).

Next Bcc.L/BSR.L in assembly list

Select this or its hotkey (Ctrl/L) to see the next relative address with a
.L extension. If you mean this as .W, then you can edit it to .W for
avoiding the problem in
                Relative Extensions
                . You should
replace .S/.L in 68000 files with .B/.W

See also Treat .L as .W in Relative addresses under

                Sundry Preferences
                .

Next Bcc.S/BSR.S in assembly list

This works like Next Bcc.L/BSR.L. (See above). Its hotkey is
Ctrl/S. It does not find .b's, only .s's, so you can change .s's to .b's.

5. Next 68881/2 Instruction in Assembly List

Select this or its hotkey (Ctrl/F) to see the next line in the assembly list
with 68881/2 syntax. Use this to see if your sc contains 68881 instructions.

Next >68000 instruction in assembly list
```

Select this or its hotkey (Ctrl/G) to see the next line in the assembly list with syntax for 68010, 68020, or 68030 but not 68000. I recommend that you always use 68000 compatible mc, unless your program requires AGA or FPU.

Next Privileged/MMU Instruction in Assembly List

Select this or its hotkey (Ctrl/P) to see the next line in the assembly list which is priveleged, or a MMU instruction, or TAS, or MOVE from SR. None of these should occur in applications programs.

Next Private XREF in Assembly List

If any assembly reports that your program contains XREF's or XDEF's, and so can be saved as object code but not mc, it may well be that the XREF's are actually misspelled _LVO's to Amiga OS3.1 library routines. So, use this option or its hotkey (Ctrl/X) to seek out each private XREF.

1.44 step

Single step mc

Select Step mc to step your program, when debugging. Its hotkey is the <space> bar.

Debugging With Tandem Important information!!

When Tandem debugs, it does not create a separate process for your program. Rather, it runs your program under the process structure that Tandem inherits from the CLI or workbench. Your program must be able to operate in that environment. And, if it can operate under either the CLI or Workbench, then it must test whether it is running under the CLI. That is to say, your program must not try to receive a Workbench startup message. If you run Tandem under the workbench, see
Workbench
.

If you intend your program to call _LVOCurrentDir, or intend in some other way to assume something about the program's AmigaDOS environment, it is important to read to read
Writing a Front End
to ensure an
appropriate debugging environment. See also
Dbug CD
.

When you run Tandem, you can if you wish specify a typical set of parameters that your program might receive when it is run under the CLI. Tandem does not use any parameters. Instead, it remembers A0 and D0 when it starts up, and places these values in A0 and D0 when you begin debugging. Thus, your program has access to the parameters you gave to Tandem, as if you had given them to your program.

You should always write your program in such a way that:

1. There is a subroutine which detects which resources the program has allocated, and frees them. (e.g. if you create memory buffers, it tests them to see if used, and frees them if so). This is useful if during the running of your program, you decide to abandon it at that point, and fix some bugs in the sc, and re-assemble. Then, you can get Tandem to jump the PC to your close down section, where it BSR's the freeing of all resources, and put a breakpoint after that BSR, and run to the breakpoint so the resources are freed, before you re-assemble. Otherwise, the resources will keep getting allocated more and more, and never de-allocated, polluting the memory.

Take great care in going to the edit window when your program is half way through executing, since if you change the sc, the mc will disappear, making it impossible to de-allocate resources. For this reason, Tandem lets you lock the sc, if debugging is in progress.

2. You should write your program if possible so it can be re-run without re-assembly. This means:

   (a) not assuming initial values in DS fields.
   (b) not writing to DC fields.

   This is not a vital point, but it gets annoying having to re-assemble before each time you start running it.

See the section on
Writing a Front End
for further details.

Your program does not use the CLI stack. Instead, Tandem puts your stack in a memory buffer, and rebuilds the stack whenever your program returns to a zero stack level. When you single step or breakpoint, your SP must be longword aligned (and in general the Amiga OS probably requires your SP to always be longword aligned - don't push .W or .B values!!). See also

Stack Usage
.

Tandem puts the stack size at the root of your stack. Then, it pops the registers and PC off the stack, whenever it has to execute. If your program RTS's to the base of its stack (i.e. it closes down), Tandem recaptures it and rebuilds the stack. If you set breakpoints, Tandem puts JMP's at each breakpoint, and if your program hits them, it recaptures, and pushes your regs to your stack. This is unlike most debuggers, which poke ILLEGAL's into your program at breakpoints, and place a trap vector. Tandem does not use processor traps, and does not look at or report the value of the SR. Also, if your program hangs up, Tandem cannot break into it, but you will simply have to reboot. Thus, Tandem is primitive compared to other debuggers but on the other hand is much easier to use.

Tandem when it returns from a single step or a run, looks for the PC location in the assembly list, and shows the PC location on the main window, with the line corresponding to the PC with a coloured background. Also, any lines containing breakpoints are printed with a coloured foreground. The current register values are displayed in their buttons. You can edit all the registers except A7, and jump the PC, by clicking them. You can also

change registers & memory by
Immediate Mode
commands.

Because Tandem uses 3-word JMP instructions to set breakpoints, it means:

1. You cannot place breakpoints 2 or 4 bytes apart (0 bytes apart is ok).
2. You cannot place a breakpoint 2 or 4 bytes before the PC.
3. (Annoyingly) you cannot place a breakpoint 2 or 4 bytes before a labelled line.

These restrictions can be annoying, but you get used to them. If you violate the above, Tandem will refuse to Run mc. You will get the following error message:

Breakpoint conflict

Tandem ignores breakpoints when single stepping, so from single stepping point of view, it doesn't matter where the breakpoints are.

If you really have to place a breakpoint just above a labelled line, all I can suggest is that you re-assemble with the line temporarily padded out with NOP's. Sorry about that.

Incidentally, you must never place a breakpoint where your program will be in supervisor mode, or where DosBase or IntuitionBase are locked, or Tandem will not be able to regain control, and the Amiga will hang up. there are certain other obvious places not to place breakpoints - e.g. where you are banging the hardware registers (shame!) &c. if Tandem has its own private screen, your programs will open on the default public screen, not on Tandem's rpivate screen.

Tandem assembles your program into a memory buffer, in object code form, i.e. with relative addresses relative to address $0000. The first time you single step or run your program, you will see on the assembly list, the relative addresses become absolute, relocated to the memory buffer your program resides in. (For that reason, Tandem will make you re-assemble a program before saving it as an mc file).

In the Tandem/Teaching dir, there are lots of .asm files, of gradually increasing complexity, for you to familiarise yourself with stepping, breakpointing and running under Tandem.

There are many ways your program can crash or hang up when single stepping or running, so always, always, save your sc before debugging!!

Single Stepping

Be sure to understand the above points before you single step or run. You can single step by selecting it on the main screen's menu, or clicking the Step mc button, or by pressing the <space> bar, which is a hotkey for single stepping.

The following restrictions apply to single stepping:

1. Tandem forbids stepping out of the range of your mc

```
2. Tandem forbids a step which would place the PC in a non-mc address
   (e.g. trying to step to a DC or DS address.

3. Tandem forbids stepping through the following:
   - priveleged instructions (including MOVE from SR even on the 68000)
   - TAS (TAS should never be used in the Amiga)
   - JMP (for technical reasons)
   - MMU instructions

Tandem treats the single stepping of a JSR in a special way, viz:

1. If Tandem is on a private screen, it places Tandem's screen behind all
   other sceens. Else, it places the main window behind all other windows.

2. it calls the JSR as a whole - it does not step through it.

3. finally, it brings Tandem's screen & main window to the front.

Tandem thus treats JSR as a "single step" rather than a subroutine, because
I assume that its use is for _LVO library ROM calls. All programs should
regard the ROM kernal as a "black box".

Despite the above restrictions, single stepping can still easily crash your
system or hang up, so make sure you save your sc before you do debugging.

If you specify
                Dbug CD
                , then Tandem will set the CD to the
dir you specify while single stepping.
```

1.45 run

```
                Run mc (After Setting Breakpoints)

n.b. you should read the
                Step mc
                 section before reading this
section.

To run mc, make sure your breakpoints are set. If your program runs until
the stack level RTS's to zero (i.e. exits), then Tandem will regain control
and it will rebuild the stack and reset A0 and D0. But if your program hits
a breakpoint, it will remain frozen there until you again run or single
step.

When you run mc, Tandem first puts the Tandem screen/window behind all
others before running, and on return it brings the Tandem screen/main window
to the front again. (Don't forget, you can cycle through the screens with the
LeftAmiga/M hotkey built into Amiga's operating system).

If you set
                Dbug CD
                , then Tandem will also set the CD to the
dir specified while running mc or single stepping.
```

1.46 svmc

```
Save mc

To save mc, select Save mc from the main window menu, or click its
button. If you have used mc for debugging, Tandem will direct you to
re-assemble.

Tandem puts up an ASL requester for file selection.

Note that the mc is saved as a runnable module; it can be run from the CLI,
or from the Workbench by creating a task icon of the same name with suffix
.info
```

1.47 llst

```
                Llist

Select or click Llist to send an assembly list to the printer. You
can first go to
                Prefs
                 if you want to specify its format.

Tandem simply sends the lines as plaintext to PRT:, so they are filtered
through your Workbench printer prefs.

Because of Tandem's user interface, I don't think you'll ever need printouts
of sc or assembly lists. This is my contribution to saving the forests.
```

1.48 calc

```
                Calc

You can use Calc to get the value of symbolic expressions. You
input an expression, which can use any of the symbols visible in the

                View sym
                 requester, which include all the Amiga OS 3.1
symbols. e.g.:

 _AbsExecBase+5

is a valid input. Tandem shows you its value in hex and decimal, and its
type (i.e. absolute/relative/MACRO label/EQUR/REG/private XREF). See

                Tandem Symbols
                 for an explanation of symbol types.

If you input a REG or EQUR or MACRO label then the value won't mean
anything.

You can also use
```

Immed
to do much the same things. But unlike
immediate mode, there is no risk of crashing the system with Calc.

1.49 sym

View Symbols

If you select View sym, Tandem puts up a special requester to allow
you to view the contents of Tandem's symbol tables. You can refer to

Tandem's Symbols
for more information about symbols.

With this requester, you click the symbol table you want, and then it comes
up. The IncAll.i symbols are so many, that the slider is not sensitive
enough to find the exact spot, so use it to get close, and then the ^ and V
buttons for fine tuning. If you click the first line of the table, you get
back to the list of tables; when you finish perusing, click OK.

1.50 imm

Immediate Mode

Tandem has a very powerful feature, normally only seen in interpreting
languages like AmigaBASIC, viz. the ability to execute mc in immediate mode.
This is extremely useful for all sorts of purposes when debugging. Of course
it also has its disadvantages - like single stepping and running, it can
crash the system.

The same restrictions apply in immediate mode as
Step mc
.
Note also the following:

1. An immediate mode line cannot be labelled.

2. You can use immediate mode to modify memory, or re-arrange the stack, or
   do all sorts of potentially useful things.

3. You can use immediate mode for BRA, BSR and the like, but only to branch
   within the assembled mc.

4. You cannot use immediate mode for pseudo ops.

5. You cannot invoke a MACRO reference in immediate mode (obviously).

6. If you JSR the label of a subroutine, then Tandem will do it as a single
   step. But if you BSR it, Tandem will step it through.

7. It is ok to JSR an _LVO to the ROM kernal, but it is your responsibility
   to load up the registers (such as A6) with appropriate values first.

8. If you use immediate mode to fiddle with your SP, Tandem does not defend itself against your mucking up your stack. Similarly if you poke to memory &c.

1.51 pc

Changing the PC Value

If you click the PC line button, Tandem will ask you to click an sc line, and it will place your PC on that line. The line you click must be executable mc, not a pseudo op or a comment.

Take care with this: if you jump into the middle of a subroutine, and then run, the RTS will not necessarily be to where you want it. Ditto for jumping into the middle of DBcc loops.

When the PC is at a line, Tandem shows its background in a differrent colour.

If you click this option, and then wish you hadn't, just press Esc and the PC will be left alone. If you have moved somewhere else in the asm list and want to get back to wherever the PC might be at, just click the "PC line" button, and then Esc, and you will see the part of the asm list containing the PC.

1.52 memy

View a Memory Block

If you select Memory, Tandem will ask you for an address to be viewed. You can input any valid symbolic expression; if you input a relative address, Tandem will relocate it relative to your program's position in memory (even if the expression is out of the range of your program).

While looking at the memory block, press up and down arrows to skip forward of back a window-full; press ok when finished.

If you put > before your expression, Tandem will treat it as an indirect address. e.g. suppose you have a DS.L labelled Bill, which happens to contain the value $10172366, and you input >Bill, then Tandem will not show you the address #Bill, but the address (Bill), i.e. $10172366.

If you put ! and a register name, e.g. !A0, then Tandem will show the memory pointed to by that register. e.g., to view your stack: !A7

You can also put !exp(Rn), e.g. !xxp_buff(A4), so that you can view elements of a structure pointed to by Rn.

You can use both > then !, e.g. >!xxp_buff(a4) to see what is at where xxp_buff(a4) points to.

If you are looking at memory, and the first longword thereof is a pointer, you can simply input > to see the address that pointer points to.

The Amiga setup you are using may not take too kindly to your asking to see the hardware regs at $DFF000, since some are write only. Also, you may get odd results or even a crash if you ask to see non-existent memory. For example, the system might show you the contents of memory that exists at some other address.

Incidentally, Tandem does not have a built-in disassembler. (The next release of Tandem will).

1.53 brax

Set or Clear a Breakpoint

You should read the
Step mc
section before reading this.

If you click one of the three breakpoints, and it is set, then Tandem will clear it.

If it is already clear, then Tandem will prompt you to click the assembly list line to contain the breakpoint. If you put it in a bad place, it will either tell you immediately, or when you try to Run mc.

When a breakpoint has been put at a line, Tandem will show its foreground in a different colour.

1.54 regs

Viewing and Changing the Registers

Tandem maintains a set of register values when you debug. These are permanently displayed on the main window. (Tandem ignores the SR and MMU registers).

See also
Stack Usage
When you first assemble your program, Tandem places the D0 and A0 ↩ values
given to it by the CLI to your program's registers, as I described under

Step mc
, so you can pass sample CLI parameters to your program. Tandem also builds a stack, with the stack size above the root (like in the CLI environment), and a return address to Tandem at the root. During debugging, if your program "exits" to this root address, Tandem re-builds the stack, and re-initialises A0 and D0. The other registers initially contain random values.

```
You can click any of the registers, except A7, to edit it; and of course you
can use
                Immed
                 to change any of them, including A7 (careful!).
To change PC you can click the
                PC line
                .

Tandem asks you for a new value for the register, and you input the register
name, then \, then any valid symbolic addess, e.g.:

  A6\_AbsExecBase


The CCR bits are also on permanent display. Click any of them to flip-flop
their value.
```

1.55 svob

```
Save an Object File

Normally, you will use save sc to save your program as an already linked
runnable file. You will therefore hardly ever want to save an object file.
Tandem always resolves all XREF's to ROM _LVO's to the ROM  kernal. In an
error free assembly, the following circumstances would prevent Tandem from
being able to save your program as a runnable file, but still allow it to
save your program as an object file:

  - the existence of XDEFs
  - the existence of private XREFs (i.e. XREFs that are not _LVOs)
  - not having an executable instruction as the entry point to the
    program, at relative address $0000

And even if none of the above apply, you can still save your program as an
object file. You can then use a linker (such as ALINK on the Fred Fish or
Aminet disks) to link your object file in with other object files.

Most users will never need to save object files.
```

1.56 svcu

```
                Save a Custom Symbol File

In the section on
                Tandem Symbols
                 you will see that Tandem
can load three files of pre-assembled symbols when it warms up, viz.:

 - incall.consts        (which contains all the OS 3.1 .i file constants)
 - ssxref.consts        (which contains _AbsExecBase and all _LVO's)
 - custom.consts        (which you can use for other pre-assembled consts)
```

When you make an assembly, you can save the symbols table to one of the above files. Tandem will remove all non-absolute symbols, NARG, and any synthetic \@ labels. Suppose for example a release of OS 3.5 comes out. You could then make new versions of incall.consts and ssxref.consts. "Tandem Symbols" tells you how to do that. (In the unlikely event that OS 3.5 comes out, I'll make up new versions and put them where you got this from).

It is quite unlikely you will ever need to use this option. I suggest you leave incall.consts and ssxref.consts alone.

1.57 mgto

Go to in the Assembly List

In the main window's menu you will see a "Go to in Asm List" menu, which allows you to jump about in the assembly list, just like in the Edit and Jottings windows.

If you select "Seek Forward/PC" or "Seek Back/PC" and cancel the requester, Tandem will place the line which the PC points to on the window. This lets you get back there if you have "lost" it. Tandem also always places the line containing the PC on the main window after you step or run mc.

1.58 clft

Click an sc Line to Edit it

If you click the left 4 characters of a line in the assembly list, Tandem will put you in the Edit window, with the cursor on the scline corresponding to what you clicked.

But if you click a line whose sc is in an include file, nothing will happen. (To peruse include files, click the "View sc" button).

1.59 dbcc

dbug CD

See

CD'ing to the PROGDIR:
for an explanation of this button.

When Tandem single steps, or Runs mc, or does an immediate mode, it first CD's to the dir you specify in "Debug CD". Otherwise, the CD will be the dir Tandem resides in.

1.60 mdul

Modules

The "Module" button is the tall thin button with "M" above "0".

If you click that, you will get a message that it is currently inoperative. Future releases of Tandem will allow division of sc and object/mc into modules, which can be linked together for debugging. I hope to get it working so modules can be C or assembler, or dis-assembled object files. Finally the modules will be saveable as a combined mc file, or separately as object files. Tandem will then be a powerful development tool.

1.61 sc

Rules for Source Code Syntax

Tandem follows exactly Motorola's rules for 68030 assembly language, with the usual extensions that have come to be fairly standard for Amiga assemblers.

Source Code Line Format

Actual op Address Format

Tandem Pseudo ops

Tandem Symbols

Tandem expressions

Relative Extensions

1.62 lfmt

Source Code Line Format

680x0 has exellent mc, but the assembly language for that mc is poorly designed. The assembly language designed by Motorola for the 68000 was a muddle, and has been partly fixed up in 68020+. However, assemblers (including Tandem) try to keep both systems going in parallel, adding to the complexity. The instruction set of the 68030 is exceedingly large and complex in itself, and the assembly language treats the tacked on bits and pieces in a somewhat messy way; so all in all this makes the writing of an assembler for the 68030 very complex and intricate.

1. Tandem sc must be stored in plaintext - in the same form ED can read.

2. Any line that begins with a * in the leftmost position is a comment

3. A line has the following fields, from left to right:

- a label field
- an opcode field
- an address field
- a comment field

any of the fields may be omitted, except that if an address field is present, there must be an opcode field.

4. Fields must be separated by 1 or more spaces. (For compatibility reasons Tandem does not strictly enforce this, or some other of the rules, but you should always observe them).

5. The label field:

- lines may optionally be labelled
- a label if present must be the leftmost field in the line
- rule 5 should be read together with rule 6, which covers local labels
- a label may only contain the characters a-z A-Z 0-9 . _
- upper and lower case are significant; fred and Fred are different labels
- a non-local label must not start with 0-9 or .
- a label may have a trailing : which is not part of the label
- a label normally starts at the leftmost character of the line; if it doesn't, then it must have a trailing :
- non-local labels must be unique within an assembly, except if the opcode is SET. Local labels must be unique within their own context.

6. Local labels:

- there are 2 formats for local labels:

  (i) a local label can have 1 or more characters all from 0-9, with a trailing $ e.g. 0$ 23$ 04$ 4$ are 4 different local labels. (Caution: some assemblers treat 04$ and 4$ as two different labels, some don't).

  (ii) alternately, local labels can be any label starting with . e.g. .fred .Fred .harry .plus are 3 different local labels. (I recommend you always use this format. The Motorola standard did not support this format, but it is universal practice).

- local labels can be used between any 2 non-local labels, and they must be unique within their own context. But a local label cannot be referenced outside the context of its delimiting non-local labels, and the same local label can be re-used. e.g.:

```
fred:                       ;a non-local label

 bra.b jack
.jim: ds.l 1                ;the line labelled jack uses this .jim
jack: move.l d0,.jim        ;a non-local label

 bra.b jill
.jim ds.l 1                 ;the line labelled jill uses this .jim
jill: move.l d0,.jim        ;a non-local label
```

Lines labelled non-locally can refer backwards but not forwards to local labels. (The Amiga OS3.1 .i header files require this).

7. Opcodes:

   if a line has an opcode field, and no label, the opcode must *NOT* start in the leftmost column. If there is a label, the label and opcode must be separated by 1 or more spaces.

8. opcodes are of three types:

   - actual ops: which correspond to mc opcodes. e.g. MOVE, ASL

   - pseudo ops: which give instructions to the assembler. There is a basic set of pseudo ops, which different assemblers have added lots more to (pseudo ops are sometimes called "assembler directives"). The commonest are DS and DC.

   - a MACRO pseudo op, which must be labelled. That label may thereafter be used in the opcode field of a line, to invoke that MACRO.

   The programmer makes up labels; but the actual ops are a fixed set to go with the 68030 mc, and the pseudo ops are a fixed list supported by the assembler. MACRO references are case significant, just as are labels; but other opcodes are case-blind - e.g. MOVE and move are the same opcode.

9. The Address field.

   if a line has an address field, it must also have an opcode field. The exact rules for the address field will depend on its opcode. In general, actual ops have about 30 different kinds of address formats, and a particular actual op may require 0-2 (or rarely 3) addresses, separated by commas. The format of pseudo ops varies widely with the individual pseudo op. A MACRO reference address field also has its own special format.

10. If an opcode is followed by an address field, the opcode and address fields must be separated by 1 or more spaces.

12. A line may optionally have as its final field a comment field. If it does, the comment must have a leading ; which denotes that the rest of the line is a comment.

13. If a comment field has any fields to its left, the comment field must be separated from the field before it by 1 or more spaces.

Since the Amiga OS3.1 Include files do not follow the above rules exactly, I allow a little latitude in the parsing of lines, so that all of the includes will assemble ok. In particular, there is not always a space between the previous field and the comment field, and the ; for a comment field is sometimes omitted. But you should always follow the above rules scrupulously!

Text books always say to allow lots of white space in your programs. I agree, though I tend to allow less white space in my own prorams than most

programmers.

1.63 afmt

Actual op Address Format

The actual ops recognised by Tandem are as per the Motorola 68030 programmer's manual (including MMU instructions) and the Motorola 68882 FPU programmer's manual.

Opcodes can have extensions as specified in the manuals. In general, opcodes can have extensions of .B or .W or .L - and some other extensions as specified below for 68882 instructions. Tandem also reconises the 68000 .S and .L extensions for relative branches - this is a messy business, and you should refer to

Relative Extensions

for a discussion of this topic.

Where the addresses allow extensions, you should always append the extensions. In general, if an extension is omitted, Tandem nearly always assumes .W, but there are a couple of commonsense exceptions.

I will now cover the standard addressing modes one by one. Registers are parsed case blind, so e.g. D1 and d1 are the same register.

A7 can be written SP

In the examples below:

```
Dn   is  D0-D7
An   is  A0-A7
Xi   is  D0-D7/A0-A7 with scale *1/*2/*4/*8
                     with extension .W/.L
                     (default *1.W)
d8   is an 8-bit displacement
d16  is a 16-bit displacement
bd   is a base displacement .W/.L default .W
     (if a backward ref & relative, default .L)
od   is an outer displacement .W/.L default .W
     (if a backward ref & relative, default .L)
Anywhere where PC occurs, ZPC may occur.
The manuals do *not* support ZAn or ZDn, and so
     Tandem does not; simply omit the An or Dn as
     per the Motorola manuals.
```

1. Data register direct Dn

2. Address register direct An

3. Address register indirect (An)

4. Address register indirect with predecrement -(An)

5. Address register indirect with postincrement (An)+

6. Address register indirect with displacement (d16,An) }d16 must be an

```
    68000 syntax also allowed                        d16(An)    }absolute exp
    n.b. if d has fwd ref & no ext, default .W
    d interp as bd if .L  (bd,An)

 7. Address register indirect with index          (d8,An,Xi) }d8 must be an
    68000 syntax also allowed                      d8(An,Xi) }absolute exp
    Either of d8 or An may be omitted
    d8 is interp as bd if .W or .L               (bd,An,Xi)
    n.b. if d has fwd ref & no ext, default .B

 8. Program counter indirect with displacement    (d16,PC)   }d16 must be a
    68000 syntax also allowed                      d16(PC)   }relative exp
    n.b. If d has fwd ref & no ext, default .W
    d interp as bd if .L  (bd,PC)

 9. Program counter indirect with index           (d8,PC,Xi) }d8 must be a
    68000 syntax also allowed                      d8(PC,Xi) }relative exp
    either of d8 or An may be omitted
    d8 is interp as bd if .W or .L               (bd,PC,Xi)
    n.b. if d8 fwd ref & no ext, default .B

10. Absolute long                                   xxx.L     }default .L
    Absolute short                                  xxx.W     }

11. Immediate                                       #xxx      }default .W

12. Memory indirect postindexed                     ([bd,An],Xi,od)
    Memory indirect preindexed                      ([bd,An,Xi],od)
    Default bd,od .W
    Any components may be omitted

13. PC memory indirect postindexed                  ([bd,PC],Xi,od)
    PC memory indirect preindexed                   ([bd,PC,Xi],od)
    Default bd,od .W
    Any components may be omitted, but if PC is
      omitted it must be replaced by ZPC.
```

Tandem takes what you specify, and optimises it, fitting it if possible into 68000 addressing mode, else into 68020+ addressing mode. The basic formats to be optimised (in heavy type above) are:

```
    (bd,An/PC,Xi)            } In which any combination of components
    ([bd,An/PC],Xi,od)       } can be omitted.
    ([bd,An/PC,Xi],od)       }
```

which are optimised into the forms

```
    (An)
    (xxx).W
    (xxx).L
    d16(An/PC)
    d8(An/PC,Xi)
    (bd,An/PC,Xi)
    ([bd,An/PC],Xi,od)
    ([bd,An/PC,Xi],od)
```

Note that if you have an expression with forward references, Tandem does not

know in the first pass whether it is .B .W or .L. So, you should give it an extension, because to re-iterate points above:

1. If in form (d,An/PC)   default d taken as .W
2. If in form (d,An/Pc,Xi) default d taken as .B

So if d was outside the above ranges, or relative, then it would in principle be assemble-able, but in practice will cause an assembly error in the second pass, since they will not fit the defaults given in the first pass. so if d in case 2 above will be relative, you should put:

```
(d.L,An/PC,Xi)
```

This necessity for Tandem to assume extensions only happens if d has forward references, (or is an XREF), which would be hardly ever in practice.

It is my belief that the new addressing modes in the post-68000 cpu's are too complex to make programs readable & readily maintainable, and should not be used. Nevertheless, Tandem supports them of course (does anyone use them?) Their main function is to make assembers hard to write. I would strongly deprecate using (Dn) as an address - this is contrary to the philosophy of 680x0 programming, and muddles up the purpose of the registers. I would also suggest you use only even addresses for .W or .L instructions, as it is quicker, & the contrary would most likely be a sign of poor program design.

1.64 psud

Tandem Pseudo Ops

Pseudo ops are also known as "assembler directives". All pseudo ops may be labelled - some must be. They have the same general syntax as actual ops, but there are variants in the address field, based on the particular purpose of each pseudo op.

I will present the pseudo ops in 2 basic sections:

Basic Pseudo ops

Extension pseudo ops
The first class are the pseudo ops recommended by Motorola, which all assemblers should support (as far as applicable to the Amiga), while the second class are extensions, which not all assemblers support. Admittedly, some are very handy, but really you should only use the first. In practice, not all assemblers do support the first class, especially not the OFFSET pseudo op (which should never be used, and surely never has been used).

Since Tandem assembles direct to memory, it is not possible to support program segmentation, so the SECTION pseudo op is not supported by Tandem. This should make no difference in practice (Just comment them out).

1.65 bpsu

```
                Basic Pseudo ops

In the syntax, you will often see [']....['] - this means, a string in
optional single or double quotes. Like generally, consecutive delimiters in
the string are interpreted as single embedded characters. The Motorola
standard is for single quotes as delimiters, but nearly all assemblers also
accept double; to maximise compatibility, use single quotes.

Program Segments
                SECTION
                  Tandem doesn't support SECTION

                IDNT
                  object file hunk name

                XREF
                  external reference or _LVO

                XDEF
                  external definition

                MASK2
                  Tandem ignores MASK2

Program Structure
                OFFSET
                  Tandem does not support OFFSET

                RORG
                  set new relative org

                CNOP
                  align mc

                IFcc
                  start conditional assembly

                ENDC
                  finish conditional assembly

                MACRO
                  start macro definition

                MEXIT
                  quit macro expansion

                ENDM
                  finish macro definition

                INCLUDE
                  .i files, and IncAll.i

                FAIL
                  cause fatal error
```

```
                END
                  end the assembly

Symbol definition
                EQU
                  assign value to label

                SET
                  assign temporary val to label

                EQUR
                  assign register name to label

                REG
                  assign register list to label

Data definition
                DC
                  define constant

                DCB
                  define constant block

                DS
                  define storage

Printed Assem List
                PAGE
                  page throw

                LIST
                  turn on listing

                NOLIST
                  (or NOL) turn off listing

                SPC
                  skip lines

                NOPAGE
                  turn off paging

                LLEN
                  set line len

                PLEN
                  set page len

                TTL
                  set program title

                NOOBJ
                  Tandem ignores NOOBJ

                FORMAT
                  Tandem ignores FORMAT
```

```
NOFORMAT
  Tandem ignores NOFORMAT
```

1.66 sect

```
SECTION     Tandem does not support SECTION
                    Divide up mc into DATA and CODE and BSS sections
```

Tandem always assembles direct to memory, and the only disadvantage of that is that it cannot divide object code into hunks to be scatter loaded. Tandem always assembles its sc into 1 big hunk, which includes everything.

Therefore if Tandem finds a SECTION statement, it cause an error.

Most other assemblers also allow the sc to mix up data and code.

If you are assembling sc with SECTION statements, just comment them out. (If there are debugging SECTIONs, they should be completely removed, as Tandem does not need debugging data, since it has access to sc direct).

1.67 idnt

```
IDNT
```

Alternate name: IDENTIFY (which I recommend you do not use)

```
Format:      IDNT [']unitname[']
```

Specifies the hunk_unit name for the sc. The default is a null string. There is no need or reason for the hunk_unit to have a name, so few programmers will use this.

1.68 xref

```
XREF

Format:    XREF item[,item,....]
```

XREF is used on the Amiga to refer to constants which are inserted by a linker. However, there is a well established set of XREFs which always have the same value, and that is the "library vector offsets". I therefore discuss XREF under the heading of library vector offsets, and the much less common alternative use for XREFs inserted by a linker.

_LVO's and _AbsExecBase

When you make library calls, or get the address of the exec library, the correct way to do it is like this example:

```
    XREF _AbsExecBase
    XREF _LVOOpenLibrary
    version: EQU 37

    MOVE.L _AbsExecBase,A6
    LEA libname,A1
    MOVEQ #version,D0
    JSR _LVOOpenLibrary(a6)
```

So, one uses XREF to get the values of _AbsExecBase, and all the _LVO's. Now Tandem has put the values pre-assembled of _AbsExecBase and all the _LVO's into a file called ssxref.consts, which it loads when it warms up. So, no linker is needed to resolve these. In fact, if you use

```
    INCLUDE 'IncAll.i'   ;see
                INCLUDE
                then it is optional to use XREF's for _AbsExecBase and the _LVO's, ↩
                    so you
```

can omit the XREF's if you want. And even if you don't INCLUDE IncAll.i, Tandem always resolves all XREF's in ssxref.consts, so you don't need a linker to link them.

XREF's Inserted by a Linker

Tandem makes all XREF items which are not in the tandem/ssxref.consts file into labels with type xref; they are then available to your program as if they were absolute-type labels, with the proviso that if used in expressions they must stand alone. e.g. you can have Fred, but not Fred+4, if Fred is an xref-type label. See

```
                Tandem Expressions
                 for details.
```

If you use XREF's inserted by a linker, then Tandem will be unable to create mc for your assembly. You will then be able to save an object file, but not do debugging, or save an mc file.

To use such XREF's, you define them, and then may refer to them, and Tandem will append data to your object code for each reference you make to the XREF, to tell the linker where to poke the values. The linker gets values by the object files giving it XDEF's which corresponding to the XREF's.

e.g. one program can have:

```
     Fred: EQU $12345678
     XDEF Fred
```

while another has:

```
     XREF Fred
     MOVE.L #Fred,D0
```

The linker will poke $12345678 into the mc for the above MOVE.L when it links the two object code files.

The "Aminet" CDs have a linker called ALINK which you can use for linking. Few users will ever use XDEF and XREF in this way, and most users of Tandem will never use object files or linkers.

```
It is the programmer's responsibility to use XREF's with consistent .B/.W/.L
(e.g. Fred above is used as .L). This means you must carefully document the
XDEF's you create in other programs. XREF's are always absolute values.
```

1.69 xdef

```
                XDEF

Alternative Names: GLOBAL PUBLIC (which I recommend you do not use)

Format:    XDEF label[,label,...]

XDEF items must be labels with absolute (not relative) values.

For the useage of XDEF, see
                XREF
                 under "XREF's Inserted by a
Linker".
```

1.70 mask2

```
MASK2

Like all Amiga assemblers, Tandem simply ignores the MASK2 pseudo op.
```

1.71 offset

```
                OFFSET     Tandem does not support OFFSET

(Although OFFSET is supposed to be a standard pseudo op, it is so poorly
conceived that assemblers never support it).

Format:     OFFSET absexp
            ....followed by only DS's......
            terminated by RORG or OFFSET or SECTION or END

Since Tandem assembles so quickly, you can use the well-written and readable
Exec/Types.i MACRO's for structures.

Here is an OFFSET example, and the Exec/Types.i equivalent.


Example:     MyStruc: EQU 0            ;an ugly way of making a structure
            offlab: OFFSET 20
            MS_Item1: DS.W 1           ;These labels get absolute Values!
            MS_Item2: DS.W 1
            MS_Item3: DS.L 1
             RORG offlab               ;restore * to value before OFFSET
            MS_SIZEOF: EQU MS_Item3+4
```

```
Which is equivalent to, using Exec/Types.i MACRO's:

             STRUCTURE MyStruc,20      ;a pretty way of making a structure
             WORD MS_Item1
             WORD MS_Item2
             LONG MS_Item3
             LABEL MS_SIZEOF

(You can also use RS/SO/FO - see
                RS/SO/FO
                , but why not use
the Exec/Types.i MACRO's?).

(see also
                INCLUDE
                 under "IncAll.i").
```

1.72 rorg

```
RORG

Function:   Jumps to a new origin.

Format:     RORG  absexp

This is equivalent to a DS, as it simply skips the relative address forward.

Example:    RORG $2000
           Bill:          ;Bill has the value $2000 (relative)

Few users will ever use RORG. Any bytes skipped will be zero filled in your
mc.
```

1.73 cnop

```
CNOP

Function:         aligns memory

Format:           CNOP absexp1,absexp2    absexp2 is 2,4,8,16
                                          absexp1 is < absexp2, usually 0

                  Not all assemblers allow absexp2 to be 16

Examples:         CNOP 0,2  ;word align *
                  CNOP 0,4  ;longword align *

CNOP first pads memory (if necessary) with null bytes to make * divisible by
absexp2. Then, if absexp1<>0, CNOP inserts absexp1 more null bytes.

The main use of CNOP is to re-align memory after a DS.B string. It also may
```

possibly make mc slightly faster to longword align before subroutines. Some structures should be longword aligned, as the Amiga ROM Kernal manuals specify. So, to be safe, always longword align before creating an instance of any structure in the Amiga OS3.1 includes.

The Amiga loader, and _LVOAllocMem &c. align your mc at an address divisible by 8. Whether it is divisible by 16 is uncertain, so there might be no point in making absexp2 = 16.

1.74 ifcc

```
                IFcc - Conditional Assembly
```

The function of the IFcc pseudo ops is to allow conditional assembly. This is a useful feature for all sorts of purposes. For example, in the Front.i program (see

```
                Front.i
```

) by setting one EQU label to various values, Front.i can be assembled as several different programs, but yet by sharing the same file, they all document each other, and are maintained and updated together, as an integrated system. Such a controlling EQU label is often called a "switch".

IFcc is also commonly used in INCLUDE's, to determine the context the .i file finds itself in. The Amiga OS3.1 INCLUDE's all begin with a test of whether they have already been assembled, using IFND, and if true, they define the undefined label to prevent re-assembling them.

```
Format:       IFcc absexp    ;absexp, or as applicable to cc
              .....
              ENDC
```

IFcc takes these forms:

```
              IFEQ absexp      ;true if absexp=0
              IFNE absexp      ;true if absexp<>0
              IFGT absexp      ;true if absexp>0
              IFGE absexp      ;true if absexp>=0
              IFLT absexp      ;true if absexp<0
              IFLE absexp      ;true if absexp<=0
              IFC  str1,str2   ;true if str1=str2
              IFNC str1,str2   ;true if str1<>str2
              IFD  label       ;true if label defined
              IFND label       ;true if label undefined
              IFMACROD  label  ;true if label defined as a MACRO
              IFMACROND label  ;true if label undefined as a MACRO
```

The last two (IFMACROD and IFMADROND) are extensions, not supported by all assemblers. IFD cannot use a MACRO label.

IFC can be used to see if a MACRO ref has a null string in one of its parameters. e.g.:

```
              IFNC '\2',''     ;true if \2 not null
              moveq #\2,d0     ;whereupon d0=#\2, else leave d0 alone
```

```
              ENDC
```

IFcc...ENDC pairs can be nested. If the number of IFcc's and ENDC's in a program does not match, there will be a fatal error. IFcc's and ENDC's are not counted when the assembler skips through a MACRO definition, and in a MACRO reference, after a MEXIT, ENDC's are counted only if they are together after the MEXIT with only comments intervening; anything else after a MEXIT other than an ENDC will cause further counting of IFcc's and ENDC's before the ENDM not to be counted.

IFcc..ENDC are omitted from the assembly list, to make it more readable.

Nesting of IFcc...ENDC pairs can be up to a depth of 10.

If an IFcc...ENDC is false, the assembler ignores whatever is between them, except for further IFcc...ENDC's, which continue nesting until the matching ENDC to the IFcc that caused the lines to be skipped is found. Also, within a false IFcc...ENDC, MACRO...ENDM is skipped, and END will cause a fatal error.

The following lines cause a program to be structured, in that certain lines can be omitted from the assembly, or otherwise restricted:

```
    MACRO...[MEXIT]...ENDM  (see
                MACRO
                )
    IFcc...ENDC
```

The rules of how these interact are complex sounding, but really a matter of common sense. The placing of IFcc's and ENDC's should be carefully planned and documented, since if you get a fatal error for too few ENDC's, it can be a pain to find; the assembler can't tell where it/they are missing from!

1.75 endc

```
               ENDC
```

Alternative name - ENDIF (which I recommend you do not use)

Function: The ENDC command delimits conditional assembly by IFcc...ENDC

Format: ENDC

See

```
               IFcc
                for details.
```

1.76 macro

MACRO

The purpose of MACRO's is to allow a set of lines to be assembled in several

places, perhaps with variants. c.f. subroutines, which are only assembled once, but called dynamically as the mc executes.

Well designed MACRO's make sc more readable. Every programmer keeps a library of useful MACRO's. MACRO or subroutine? Experience helps decide. Often, the registers of a subroutine will be filled with a MACRO.

Format: label MACRO

A MACRO must be labelled. The label becomes the name of the MACRO, and to invoke the MACRO the MACRO name is used as the opcode.

First, must come the MACRO definition. This has the following structure:

```
macname: MACRO
 .....
 ENDM
```

the assembler does not assemble the MACRO definition; it simply skips it, until it finds an ENDM. Nothing else, not even IFcc..ENDC are noticed. If an END is noticed, it will cause a fatal error, and likewise if a MACRO is found (i.e. MACRO definitions cannot be nested), or if no ENDM is ever found. The ENDM line must not contain parameters.

Then, sprinkled throughout the program, come MACRO references, with optional parameters in the address field, like this:

```
macname [param1[,param2...
```

which causes the MACRO definition to be assembled in its entirety, each time that the MACRO is referenced. Although MACRO definitions cannot be nested, MACRO references can be nested, up to a nesting depth of 10. MACRO definitions can contain MACRO references, but of course 2 MACROs cannot reference each other, or you'll get an infinite chain, and a fatal error will occur (MACRO nesting depth exceeds 10).

Here and there in a MACRO definition, there can appear the following:

```
\n      where n is 1 to 9.
```

These invoke the parameters from the MACRO reference. e.g. consider this:

```
Fred: MACRO
 moveq #\1,d0
 moveq #\2,d1
 ENDM
.....

 Fred 123,246
```

will cause:

```
 moveq #123,d0
 moveq #246,d1
```

to be assembled. The parameters of a MACRO are considered to be strings not values, and the assembler will "expand" each line of the MACRO by

substituting the parameters for the \n's it finds. If there are no parameters, or not enough, the \n's will be replaced by nothing. The reference can contain null strings like this:

```
        Fred ,abc,,def,   which means:   \1=''
                                         \2='abc'
                                         \3=''
                                         \4='def'
                                         \5,\6,\7,\8,\9=''
```

The parameters can produce opcode, and labels, (usually) addresses, in the MACRO expansion. The MACRO definition can also use \n in its nested calls, to pass parameters to nested MACRO references.

If a parameter contains a space, it must be enclosed in < >. Even ' or " or ; are simply part of the parameter, if they are in a < > pair.

Single or double quotes in a parameter form part of the parameter.

There are 2 special parameters:

\0, which is the extension of the MACRO definition.
    e.g. if the reference is Fred.W, then \0 is 'W'.
    Unfortunately some assemblers would make \0 to be '.W' instead of 'W'.
    This is easy to notice if porting, since an assembly error results.

\@, which is used to create unique labels. Tandem maintains a string
    with an inital value _000
    If a MACRO contains any \@'s, all the \@'s in that MACRO are replaced
    by that string, and after ENDM the _000 is bumped to _001, _002, &c.
    The \@ can be used anywhere, but normally for labels.
    (the traditional Metacomco assembler used .000, but since . has come to
    be used for local lables, most assemblers including Tandem now use _000)
    Labels made this way are called "synthetic" labels. Synthetic labels can
    be referenced from outside the MACRO, but that is bad practice.

Tandem also maintains a special variable called NARG, which outside of MACRO references is zero, but inside the MACRO has a value equal to the number of parameters (including null strings) of the invoking reference. e.g.:

```
    Fred 1,2,,4
```

would cause NARG=4. You can then use NARG in expressions, normally in IFcc, e.g.

```
   IFGE NARG-4
   moveq #\4,d0  ;only assemble this if \4 exists (but could be null!)
   ENDC
```

There is a further pseudo op, MEXIT, which can cause the MACRO to stop assembling before the ENDM is reached. It is usually used in conjunction with IFcc...ENDC. e.g.:

```
   IFLT NARG-2
   MEXIT        ;stop expanding if NARG<2
   ENDC
   ....
```

```
   ENDM

Nothing will be seen between MEXIT and ENDM, except 1 or more consecutive
ENDCs immediately after the MEXIT, with nothing interposing except comments.
So for example you can't have:

   IFLT NARG-5  ;only assemble if  2<NARG<5
   IFGT NARG-2
   MEXIT
   ENDC
   NOP           <- this NOP will stop Tandem looking for any more ENDCs
   ENDC          <- this ENDC will not be seen, causing a fatal error
   .....
   ENDM

But ok is:

   IFLT NARG-5  ;only assemble if  2<NARG<5
   IFGT NARG-2
   MEXIT
   ENDC
   ENDC          <- this ENDC will be seen ok since it is consecutive
   .....
   ENDM

MACRO definitions are omitted from the assembly list, to make it more
readable.

To learn to understand MACRO's, look at the MACRO's in the Amiga OS3.1
Exec/Types.i, and see how they work. The BITDEF MACRO is expecially clever.
If you can understand the BITDEF MACRO, just remember that nobody likes a
clever Dick.
```

1.77 mexit

```
                MEXIT

MEXIT causes a MACRO reference to stop before reaching its ENDM, usually in
conjunction with IFcc..ENDC

Format:        MEXIT

For details (especially about ENDC after MEXIT) see
                MACRO
                ).
```

1.78 endm

```
                ENDM

Function:      Finishes a MACRO definition
```

```
Format:        ENDM

For details see
                MACRO
                .
```

1.79 incl

```
                INCLUDE

INCLUDE is used for including a diskfile as if it were part of the sc. Most
assemblers read bits of the file at a time into memory twice through. But
because of Tandem's method of debugging, it must be loaded into a memory
buffer. Tandem always has loaded into a buffer a file IncAll.i, and other
INCLUDE files must be loaded as they are found in INCLUDE statements. After
that they stay there - so, even if you assemble a new sc file, perhaps with
completely different INCLUDE's, the old INCLUDE's stay there. To flush the
INCLUDE buffer of all except IncAll.i, select "Mem buffs" on the Main
window, and then select "Flush INCLUDE buffer". You can click "View sc" on
the Main window, to see which INCLUDE files are in the Includes Buffer, and
to peruse them if you wish. You cannot directly edit the contents of the
INCLUDE buffer. You should only make well-debugged things into .i files.

If the Include Buffer overflows during assembly, or if one of the INCLUDE
files you specify is unloadable, you get a fatal error. INCLUDE files may
include other INCLUDE files, down to a depth of 10. If the nesting depth
exceeds 10 (which would probably be caused by 2 INCLUDE files tying to
INCLUDE each other) you also get a fatal error.

It is customary to give the suffix .i to INCLUDE files.

There is nothing special about INCLUDE files. They contain sc just like the
sc in the Edit window. Here's how it works:

1. your program contains:

     INCLUDE ['][path]file.i[']            e.g.  INCLUDE 'IncAll.i'

2. When Tandem boots up, like most programs it does a CD to its PROGDIR:
   (Such programs must always CD back to the original CD before they exit).

   After that, Tandem adds Tanden/Includes to the assigns of the INC:
   directory, creating INC: if it doesn't already exist.

   Such a directory, from the earliest days of the Amiga, should only be
   for include files, just as LIBS: is used (compulsorily) for libraries.

3. When Tandem encounters your INCLUDE pseudo op in an assembly, it scans
   the Includes Buffer for a file with the same path, relative to INC:, as
   file.i, and if so, starts getting subsequent lines from there. If not,
   it loads the file, and continues.

   Note that Tandem will only find the file already in the Includes buffer,
   if its path is expressed exactly the same, letter for letter, as
   it was when it was loaded before (although case is not significant). So:
```

if it was loaded in a previous assembly as:

INC:MyInc.i

and the second time as:

inc:myinc.i

then it will be found already in the Includes buffer. But if the second time it is included as:

work:tandem/includes/myinc.i

then since the path is not identical letter for letter, Tandem will not recognise that it is already in the Includes buffer, and will load it again (if it will fit).

You can avoid this by flushing the Includes buffer, if you start to work on a different project which might re-use the same includes in a different path format. To flush the Includes buffer, select "Mem buffs" from the Main window.

4. See
INCDIR
for more discussion of how Tandem looks for INCLUDE files.

5. When Tandem comes to the end of the INCLUDE file, it pops back to where it was before the INCLUDE statement was found.

6. It is customary to put self-contained subroutines and constant blocks and MACRO's in INCLUDE files but this is not essential. Theoretically you could have half a MACRO in the include file, and the matching ENDM in the main file, though that would be poor practice.

7. In the Amiga OS3.1 .i files, you will see that they conditionally include each other, in an integrated environment of MACRO definitions and EQU statements, setting up many constants for use by your programs.

IncAll.i

INCLUDE 'IncAll.i'

is a special case, unique to Tandem.

If you inspect Tandem/IncAll.i, you will see that IncAll.i is not a terribly long file, but it contains all the MACRO definitions in the Amiga OS3.1 .i files. But, if you INCLUDE IncAll.i, Tandem also gives you instant pre-assembled access to all the constants defined by all the OS3.1 .i files, and to all the _LVO's. You need do no further INCLUDEing or XREF'ing - they are all there! (Tandem/Support/tanlib.i is also included in IncAll.i).

The only disadvantage is that you must not duplicate any of those labels. Suppose for example you create a line labelled Menu: - that would cause an error, since the OS3.1 .i files have already created a variable by that

```
name. (tanlib.i labels all start with TL or xxp_). Actually I think this is
an advantage, not a disadvantage, as it causes all OS3.1 labels to be
"reserved".

When Tandem loads itself, it loads 2 files: Support/incall.consts and
Support/ssxref.consts, which contain all the information needed.

You can save all the absolute labels from any assembly you do if you like,
in a file called 'custom.consts', which Tandem also tries to load and makes
available to IncAll.i   You can select 'Save symbol table (rare)' in the
menu of Tandem's Main window to do that. I suggest you don't fiddle with the
incall.consts or ssxref.consts or IncAll.i files.

For further information about IncAll.i, see
                Tandem Symbols
                .

Front.i

Tandem has a special set of files:

   Tandem/Includes/Front.i
   Tandem/tandem.library

See
                Front.i
                 for what these are all about. You can use them
for your own projects if you want to.

Your own .i Files

Over time, you will come to write your own subroutines and MACRO's, which
you want to make general use of. You can as you write these and get them
well debugged, put them in .i files in the Tandem/Includes dir.

.i files should only use labels you want to call from outside the scope of
the file. Internal labels should be local labels, so you don't create a
whole lot of unwanted non-local labels in assemblies that use your .i files.
```

1.80 fail

```
FAIL

Format:       FAIL

FAIL causes a fatal error. It would genrally be used in IFcc...ENDC's. e.g.:

              INCLUDE 'Fred.i'     ;get Fred.i (contains Fred.version)
              IFLT Fred.version-5  ;check Fred.version at least 5
              FAIL                 ;else quit
              ENDC
```

1.81 end

```
END       Compulsory in some assemblers
```

END stops the assembly. Any further lines after END are ignored. Some assemblers (such as A68K) make it compulsory to have an END.

END (or the physical end of lines if there is no END) must not occur within a MACRO...ENDM, or when there have been more IFcc's than ENDC's, or you'll get a fatal error.

1.82 equ

```
EQU

Function:   EQU is used to create labels with absolute values.
            (It is also allowed but unusual to give relative values)

Format:     label EQU absexp   ;(relexp also permitted)
```

An EQU must be labelled. All the "magic numbers" your program uses should all be defined by EQU's near its start, which numbers can then be referred to by meaningful labels.

```
examples:     k1:    equ $1234
              pcode: equ 3241
              k2:    equ k1+3
```

For compatibility with some other assemblers, Tandem allows you to replace EQU by = but I don't recommend this, as it reduces compatibility overall.

It is also permissible to assign relative values by EQU, e.g.:

```
            Fred: NOP
            Bill: EQU Fred
```

But this is poor practice, and some assemblers may not permit it.

1.83 set

```
SET

Function:  SET is used to create labels with absolute values.
           (SET can also assign relative values, but this is unusual)

Format:    label SET absexp
```

A SET must be labelled.

A SET is the only label that can ever have its value changed (a local label cannot have its value changed within its own context, unless it is a SET - but outside its context, its name may be re-used, since it no longer

```
exists).

e.g.:       Fred: EQU $1234
             MOVE.W #Fred,D0
            Fred: EQU $4321   ;not allowed - Fred cannot be re-defined
             MOVE.W #Fred,D1

            Bill: SET $1234
             MOVE.W #Bill,D0
            Bill: SET $4321   ;ok, since a SET can be re-defined
             MOVE.W #Bill,D1
```

1.84 equr

```
EQUR

Function:  EQUR is used to give a name to a register, from D0 to D7 or A0 to
               A7.

Format:    label EQUR reg

Example:   libptr: EQUR A6

An EQUR must be labelled.

This pseudo op is rarely used. Anywhere D0-D7 or A0-A7 can be used, an EQUR
label of that register can be used. e.g.:

            libptr: EQUR A6
             ....
             MOVE.L _AbsExecBase,libptr   ;same as Move.l _AbsExecBase,A6
```

1.85 reg

```
REG

Function:  REG is used to give a name to a MOVEM reglist.

Format:    label REG reglist

Example:   allsv: REG D0-D7/A0-A6

This pseudo op is rarely used. It applies only to MOVEM. e.g.:

            allsv: REG D0-D7/A0-A6
            ....
            MOVEM.L allsv,-(A7)
```

1.86 dc

DC

DC is used to define constants.

The DC is the only pseudo op that can make forward references in its expressions. e.g.:

```
Fred: EQU Bill  ;not allowed, since Bill is a forward reference
Bill: EQU 2

Fred: DC.L Bill ;allowed, since DC may make forward refs
.....
Bill: DS.L 1
```

A forward referance in a DC may be to an absolute or relative label, or even to an XREF.

It is my opinion that:

1. Programs should not write to DC's or DCB's.
2. Programs should not assume initial values for DS's.

However, not everyone agrees with me about this.

```
Format:    DC[.i] exp[,exp...]      .i is .B .W or .L default .W
                                    I recommend you never omit .W
                                    (but see "DC for 68881" below)
                                    (but see also "DC.B Strings" below)
```

it is customary to label a DC (or the first of a group of DC's), but not compulsory.

```
Examples:  jack: EQU $12
           jill: EQU $1234
           fred: EQU $12345678
           june:
            DC.B $21,'A',$3F,jack
            DC.W $4321,'AB',$AB,jill
            DC.L $87654321,'ABCD',$BADFACED,fred,june
```

In the above, you will see that a DC.L can have relative expressions in its list. You should refer to

Tandem Expressions
for rules

about expressions and the ranges allowed for .B .W and .L

If you use DC.W or DC.L Tandem aligns the mc to an even address before inserting the MC (and if the DC is labelled it gets that even address of course).

It is customary to put DS.W 0 after a DC.B, to re-align the mc if required.

DC.B Strings

A DC.B list can contain strings in its list, like this:

```
          strings: dc.b 0
           dc.b 'string 1',0 ;1
           dc.b 'string 2',0 ;2
           ds.w 0
```

If your programs write to DC's, or read from DS's before writing to them you must re-assemble before re-running or re-stepping when debugging.

DC for 68881/2 (FPU)

As an extension of DC to allow you to send floating point values to the 68881/2, you can use DC.P, for packed decimal strings. These are put in as constants - not expressions - in decimal or decimal floating point format, like this:

```
fvals: DC.P 1234,-678.12,3.5678E+17,-4.71E-20,0 ;FPU data
```

You can also use other FPU extensions for DC - e.g.:

```
fsings: DC.S -123,467.81
fdoubs: DC.D 3.14159265358979,2.7182818
fexes:  DC.X 123.45,-63.89E-79
```

But - be warned: If you use Tandem for DC.S, DC.D, or DC.X statements, Tandem will use the 68881/2 to assemble them - so if your computer does not have a 68881/2 and you use DS.S, DS.D, or DS.X, the Amiga will crash.

But computers without a 68881/2 can assemble DC.P ok.

Incidentally, the same applies with # constants in FPU actual ops. e.g. you can have:

```
   FMOVE.P #61.89E-600                          } Only floating point
                                                } constants (not express-
   FMOVE.S #123.45,D1        } Only Amigas with  } ions) can be used
   FMOVE.D #2.7182818,-(a7)  } a 68881/2 can     } after # with extension
   FMOVE.X #-24.62E16,FP3    } assemble these    } .P .S .D .X
```

Rules for Floating Point Constants

Tandem converts your floating point constant to a normalised packed decimal, and then if it is not .P, uses the 68881 to convert it to .S .D or .X

The mantissa can have any number of digits before or after the decimal point, and the exponent if present is E or e followed by - or an optional + and then 1-3 digits of exponent. Tandem normalises your input, and the first 17 non-zero digits of the mantissa (if any) are significant. There need not be a decimal point or a leading + or - in the mantissa, but there must be at least 1 digit.

If there are more than 17 digits in the mantissa after any leading zeroes, they are ignored, but if to the left of actual or implied the decimal point they still contribute to the exponent.

```
The exponent, after it is normalised, must be from -999 to +999.

Examples of good floating point constants:

1.2345E+2    assembles as       0002 0001 2345 0000 0000 0000
-123.45e+2                      8002 0001 2345 0000 0000 0000
1.2345E-2                       4002 0001 2345 0000 0000 0000
-1.2345E-2                      C002 0001 2345 0000 0000 0000
.000376421                      4004 0003 7642 1000 0000 0000 (normalised)
123456789123456789777           0021 0001 2345 6789 1234 5678 (truncated)
17E20                           0021 0001 7000 0000 0000 0000 (normalised)
3.14159E-721                    4721 0003 1415 9000 0000 0000
-0.0                            0000 0000 0000 0000 0000 0000 (0 always +)

Examples of bad floating point constants:

0.0.0    only 1 . premitted
E67      no digits in mantissa
23E      must be 1-3 digits in exponent if present
6E1000   exponent must be -999 to +999
23E999   as normalised, exponent is 1000, so bad since > 999
```

1.87 dcb

```
DCB

Alternative names:  DSB  BLK   which I recommend you do not use

Function:  Define a constant block

Format:    DCB[.i] absexp,exp    pokes exp into mc, absexp times.
                                 .i can be .B .W or .L default .W

Example:   DCB.B 5,'A' ;equivalent to  DC.B 'AAAAA'

Like DC.W and DC.L, DCB.W and DCB.L word align the mc before poking.

DCB is equivalent to DS (but see DS about that). If exp<>0, you should use
DCB not DS, to ensure maximum compatibility between assemblers.

If your programs write to DC's, or read from DS's before writing to them
you must re-assemble before re-running or re-stepping when debugging.
```

1.88 ds

```
DS

function:       Defines (variable) memory Space

format:         DS[.i] absexp1[,absexp2]   .i is .B .W .L default .W
                                           skips absexp1 bytes
                                           fills wifth absexp2, default 0
```

DS is used for the equivalent of variables in higher level languages. It is normal but not compulsory to label DS's (except when DS.W 0 is used for alignment).

Tacking on absexp2 is an extension - better to use DCB for compatibility. absexp2 is nearly always 0, and should then be omitted.

If your programs write to DC's, or read from DS's before writing to them you must re-assemble before re-running or re-stepping when debugging.

In my opinion, programs should not write to DC's, or assume initial values for DS's. But not all programmers agree.

DS.W and DS.L word align mc before poking; their label if any is of course given its value after such aligning. It is common to use DS.W 0 instead of CNOP 0,2 to word align mc. e.g.:

```
  DC.B 'graphics.library',0
  DS.W 0
```

It is advisable to always put a DS.W 0 after a series of 1 or more DS.B statements. Note that DS.L only word aligns, it does not longword align.

1.89 page

PAGE

Function: Send a form feed when doing an assembly list.

Format: PAGE

a PAGE statement is not printed in an assembly list sent to the printer; instead, Tandem sends a form feed to the printer.

1.90 list

```
               LIST

Format:       NOL[IST]
              ....
              LIST
```

When Tandem sends an assembly list to the printer, it always suppresses the above statements from the assembly list. It will also suppress the listing of all lines within a LIST...NOLIST pair. Superfluous LIST's and NOL's will be ignored (they don't nest).

On the Main window, LIST and NOL have no effect, and are always shown, along with the lines between them.

```
You can also select
                Prefs
                 from the Main window, to
suppress the printing of MACRO expansions and/or INCLUDE's, if you wish,
which over-rides NOL...LIST's as regards MACRO expansions and INCLUDE's.
```

1.91 nol

```
                NOLIST (NOL)

Format:        NOLIST
               NOL

For the function of this, see
                LIST
```

1.92 spc

```
SPC

Function:    Send blank lines when printing an assembly list

Format:      SPC [absexp]                default absexp 1

When printing an Assembly list, Tandem does not print SPC statements, but
sends absexp line feeds to the printer.

Note that blank sc lines still have the relative address at their left in an
assembly list, unlike SPC which produces completely blank lines.
```

1.93 nopage

```
NOPAGE

When printing an assembly list, Tandem does not print a NOPAGE statement;
instead, it causes Tandem to ignore the operative PLEN, and never send form
feeds to the printer.

A PAGE will cancel a NOPAGE and will turn on paging again, starting right
away with a form feed.
```

1.94 llen

```
                LLEN

Function:    specify maximum characters in a line
```

```
Format:      LLEN absexp             (absexp is 60 to 132)

When Tandem begins an assembly, it sets a llen variable as per
your
                Prefs
                , set from the Main window. By default, this is 70.

If Tandem finds an LLEN, it puts the absexp into the llen variable,
over-riding your "Sundry" preference.

All lines printed in an assembly list will have a length not greater than
the llen variable. LLEN statements are not printed in the assembly list.
```

1.95 plen

```
                PLEN

Function:    specify maximum lines to a page

Format:      PLEN absexp             (absexp is 24 to 100)

When Tandem begins an assembly, it sets a plen variable as per
your
                Sundry
                 preferences, set from the Main window. By
default, this is 72.

If Tandem finds an PLEN, it puts the absexp into the plen variable,
over-riding your "Sundry" preference.

All pages printed in an assembly list will have a length not greater than
the plen variable. PLEN statements are not printed in the assembly list.

See also
                PAGE
                 and
                NOPAGE
                , which over-ride
the operative PLEN.
```

1.96 ttl

```
TTL

Function:    specify the title of an assembly list

Format:      TTL [']string[']

If the string has spaces, it must be enclosed in quotes.

If Tandem finds a TTL in an assmbly, it will print the TTL string first when
it does an assembly list, as a title, else it has no title. Tandem does not
```

print the TTL statement.

1.97 noobj

NOOBJ (Tandem ignores a NOOBJ statement)

Function: cause an assembly to produce an assembly list but no mc.

Format: NOOBJ

Since Tandem assembles direct to memory, and not direct to a file, the NOOBJ pseudo op has no affect, and is simply ignored.

1.98 format

FORMAT NOFORMAT

Like all Amiga assemblers, Tandem simply ignores FORMAT and NOFORMAT if it finds them.

1.99 epsu

Extension Pseudo ops

I did not invent any of these mainly useless extensions. Here is a list of all the extensions which I found by perusing several other assemblers, and whether or not Tandem supports them. Use of nearly all of them is strongly discouraged, since they reduce compatibility.

```
 * = Pseudo ops not supported by Tandem (click to see why, &c)
(*)= Pseudo ops ignored by Tandem  (not supported, but no error)
 % = Supported, but use discouraged


      ADDSYM
       *  debugging data not applicable to Tandem

      ALIGN
       %  alternative to CNOP

      ASCII
       %  alternative to DC.B for strings

      BDEBUGARG
       *  debugging data not applicable to Tandem

      BITSTREAM
       %  alternative to DC.B %...,%...,

      BLK
```

```
 %  same as DCB

BOPT
 *  sundry program switches

BSS
 *  same as SECTION BSS

CARGS
 %  define a list of offsets

CMACRO
 *  a MACRO which can be referenced case blind

CODE
 *  same as SECTION CODE

CSEG
 *  same as SECTION CODE

CSTRING
    alternative to DC.B for C strings

DATA
 *  same as SECTION DATA

DB/UB &c
 %  alternatives to DC - Tandem treats as DC's

DEBUG
 *  debugging data not applicable to Tandem

DOSCMD
 *  execute an AmigaDOS command (why?)

DSB
 %  same as DCB

DSBIN
 *  leave a gap in mc equal in size to a file

DSEG
 *  same as SECTION DATA

DSTRING
 *  insert system date in mc, but not sc (why?)

IDENTIFY
 %  same as IDNT

ELSE
 *  extension to IFcc...ENDC

ELSEIF
 *  extension to IFcc...ENDC

ENDIF
```

```
 %  same as ENDC

EVEN
 %  alternative to CNOP

EXEOBJ
(*) save mc as object - effectively ignored

FILECOM
 *  attach comment to mc file

FO RS SO
 %  alternative for Exec/Types.i MACRO's

GLOBAL
 %  same as XDEF

IBYTES
    insert a file into the mc

INCBIN
    insert a file into the mc

INCDIR
 %  assigns a dir to INC:

INCPATH
 %  assigns a dir to INC:

ISTRING
 *  DC.B alt for strings w. bit 7 set in last byte

LINKOBJ
(*) save mc as object - effectively ignored

LISTFILE
    specify file to send assembly list

MC68xxx
    specify whether 68020,68030,68881 etc. allowed

OBJFILE
    specify mc filename

ODD
 %  alternative to CNOP

ORG
 *  assemble with a fixed origin (why?)

OUTPUT
    specify mc filename

PAD
 *  alternative to CNOP

PRINTX
```

```
 *  print to _LVOOutput (why?)

PSTRING
    alternative to DC.B for BCPL strings

PUBLIC
 %  same as XDEF

PURE
 *  set pure bit in mc file

QUAD
 %  alternative to CNOP

REPEAT
 *  (also written REPT) alternative to MACRO

RS SO FO
 %  alternative to Exec/Types.i MACRO's

SMALLDATA
 *  a system using a register as a pointer

SPRINTX
 *  DC with C formatting

SUPER
 *  suppress warnings for supervisor mode

SYM
 *  debugging data not applicable to Tandem

TRASHREG
 *  register for SMALLDATA
```

1.100 code

```
CODE CSEG DATA DSEG BSS     Tandem does not support these

These are variants of the
SECTION
 statement, and since Tandem
does not support that, it does not support these.
```

1.101 dbug

```
DEBUG BDBUGARG SYM ADDSYM        NOT supported by Tandem!

These items are used by some other assemblers to try to create symbolic
debugging environments. Since Tandem debugging has instant automatic access
to the assembly list, there is no need for these, and Tandem does not
support them.
```

Where these are followed by lines of debugging information, presumably every
line will cause an assembly error, so surround these with false IFcc..ENDC

1.102 trsh

```
                SMALLDATA TRASHREG    NOT supported by Tandem!
```

These ill-conceived pseudo ops are designed by some assemblers to create a
register to point to everything, like

```
                Front.i
                 uses A4. In
```

my opinion, this only causes a mess, using the assembler to do the
programming, as if it were a compiler. So, Tandem does not support these.

Also, systems like this destroy compatibility between assemblers, and make
assembly language harder to read.

Programs that use these would have to be re-written extensively for an
assembler that doesn't recognise them.

1.103 mc68

```
                MC68000 MC68010 MC68020 MC68030    Not all essemblers support  ↩
                   these
MC68881 MC68881 MC68851

Format:    MC68000     (etc.)
```

Tandem sets an MC variable according to

```
                Prefs
                 on the Main
```

window. During assembly, Tandem over-rides this if any MC68xxx pseudo ops
are found. But, Tandem otherwise ignores the MC variable, and simply
assembles all valid 68030, FPU and MMU instructions.

After assembly, you can scan for MC violations, &c by selecting from the
Main window

```
                Errs &c
                .
```

If MC68040 or MC68060 are found, it will cause an error. Tandem can only do
up to 68030.

1.104 opfl

```
                OUTPUT OBJFILE      n.b. not all assemblers support this pseudo op

Format:  OUTPUT [']filename[']    } The filename should preferably be in
         OBJFILE [']filename[']   } quotes, compulsorily if it has a space.
```

```
Example: OUTPUT "myfile"

When Tandem begins an assembly, it sets a default filename for your mc as
follows:

 - if the last save to sc has suffix .asm, it uses that filename without
   the suffix (& uses the same with suffix .o for object file, if any).
 - else, it leaves the mc filename alone

But if an OUTPUT/OBJFILE is found, the above is superseded. Note that if you
click
                Save mc
                 an ASL requester always comes up; the operative
default filename becomes the ASL prompt.

Tandem puts the above into the filepart of the mc file prompt; there is also
a prompt for the dir (drawer) part, which is initially null and unaffected
by the above. There is only 30 bytes available for the file part, so if you
put the complete path in OUTPUT/OBJFILE, it may be truncated. All in all, I
would discourage the use of OUTPUT/OBJFILE, as it isn't much use, and
reduces compatibility.
```

1.105 eobj

```
EXEOBJ LINKOBJ        (Tandem always does these, present or not)

Format:   EXEOBJ              not all assemblers support these pseudo ops
          LINKOBJ

These request the assembler to create an mc file rather than an object code
file.

These are effectively ignored by Tandem, since the user chooses whether to
save an assembly as mc or object (nearly always mc, of course).
```

1.106 org

```
ORG

Format:     ORG absexp         Tandem does NOT support this pseudo op

Example:    ORG $20000000

This pseudo op causes mc to be assembled, relative to a fixed address. The
only purpose of this might be to assemble mc for burning a ROM(?). Tandem
does not support this evil non-Amiga-ish pseudo op.
```

1.107 cargs

```
                CARGS    Not all assemblers support this pseudo op

Format:    CARGS [#offset,][sym.i][,...]        default #offset = #0
           .i is .B .W or .L default .W

Example:   MyStruc: EQU 0
            CARGS #20,MS_Item1.W,MS_Item2.W,MS_Item3.L

   equivalent to   STRUCTURE MyStruc,20
                   WORD MS_Item1
                   WORD MS_Item2
                   LONG MS_Item3
```

The above explains the use of this. The Motorola-sanctioned method is the use of an

```
                OFFSET
                 section, but seeing that many assemblers
```

do not support OFFSET, you might was sell use this as OFFSET. However, since Tandem has the

```
                INCLUDE
                 'IncAll.i'
```

option, why not use that, and then you can use the much more readable Amiga OS3.1 exec/types.i format, like in "equivalent to" above? Tandem assembles so fast, there is no problem about the slowing down due to MACRO references.

```
                RS/SO &c
                 is yet another method of doing offsets. This shows
```

the problem with using non-standard things; they just keep proliferating, spoiling compatibility between assemblers.

1.108 sofo

```
RS RSSET SETRS RSRESET RESETRS CLRRS RSVAL
SO SOSET SETSO SORESET RESETSO CLRSO SOVAL
FO FOSET SETFO FORESET RESETFO CLRFO FOVAL
```

n.b. not all assemblers support these, and those that do, support them in muddled up ways. The RS things are supported by some, and SO by others; Tandem does them interchangeably. FO is a backward form of SO supported by assemblers that use SO

```
Format:    RSRESET       sets rs/so counter to 0       n.b. the same
           RESETRS       sets rs/so counter to 0       counter is used for
           SORESET       sets rs/so counter to 0       RS and SO formats, if
           RESETSO       sets rs/so counter to 0       you mix them up, no
           CLRRS         sets rs/so counter to 0       probelm. Better still
           CLRSO         sets rs/so counter to 0       use exec/types.i
           RSSET absexp  sets rs/so counter to absexp
           SETRS absexp  sets rs/so counter to absexp
           RSVAL absexp  sets rs/so counter to absexp
```

```
          SETSO absexp  sets rs/so counter to absexp
          SOSET absexp  sets rs/so counter to absexp
          SOVAL absexp  sets rs/so counter to absexp
     label RS[.i] count  } these define label to the rs/so counter
     label SO[.i] count  } then bump counter by count*1,2 or 4 (if .B/.W/.L)
                         } default .i is .W
          FORESET       sets fo counter to 0
          RESETFO       sets fo counter to 0
          CLRFO         sets fo counter to 0
          SETFO absexp  sets fo counter to absexp
          FOSET absexp  sets fo counter to absexp
          FOVAL absexp  sets fo counter to absexp
     label FO[.i] count  } this defines label to the fo counter
                         } then decs counter by count*1,2 or 4 (if .B/.W/.L)
                         } default .i is .W
```

1.109 pad

```
ALIGN PAD QUAD EVEN ODD       Not all essemblers support these
                              Tandem does not support PAD
```

These are alternatives to the standard pseudo op CNOP, and illustrate how using non-standard pseudo ops simply multiplies useless alternatives. Tandem supports them, but why not use CNOP?

```
Format:     ALIGN absexp     equivalent to  CNOP 0,absexp  (absexp=2,4,8,16)
            QUAD             equivalent to  CNOP 0,16
            EVEN             equivalent to  CNOP 0,2
            ODD              equivalent to  CNOP 1,2
            PAD absexp1,absexp2  equivalent to  CNOP 0,absexp1 and fills
                         gaps (if any) with absexp2's instead of $0's) (why?)
```

I repeat that aligning 16 makes no sense, since Amiga's loader only promises to align 8.

I must admit I'm tempted to use EVEN instead of CNOP 0,2. But I would still use a MACRO like this, to ensure compatibility:

```
            EVEN: MACRO
             CNOP 0,2
             ENDM
```

When Tandem simply ignores my MACRO, but an assembler that does not support EVEN would still assemble ok. This ensures compatibility.

1.110 db

```
                DB DW DL          These are useless alternatives to DC, for DC's
UB UW UL          with particular signs, &c. If Tandem finds them,
SB SW SL          it treats them as
                DC
                . Why do people
```

```
PB PW PL          writing assemblers keep fiddling with extra
NB NW NL          non-standard pseudo ops, spoiling compatibility?
```

1.111 asci

```
ASCII CSTRING PSTRING ISTRING   Not all assemblers support these
                                 Tandem does not support ISTRING

These are extensions to DC.B with strings. They are useful for enhanced
readability, but reduce compatibility with other assemblers.

ASCII    is the same as DC.B, but contains only strings
CSTRING  CSTRING 'str1','str2'...  =   DC.B 'str1',0,'str2',0...
PSTRING  has format like DC.B, but puts the length of each string
                 in fromt of it (i.e. BCPL strings)
ISTRING  has format like DC.B, but sets bit 7 of the last byte in
                 each string. Tandem does not support ISTRING
```

1.112 bits

```
BITSTREAM    Not all assemblers support this

Format:  BITSTREAM [']nnnnnnnn[']      where nnnn are 0's and 1's
                                       If the digits are not a multiple of
                                          8, the last byte is 0 filled

         This is a useless alternative to DC.B followed by %nnnnnnnn, ...
         which is more readable and compatible.
```

1.113 spri

```
SPRINTX      Tandem does not support this

SPRINTX is a useless attempt by some assemblers to implement C-style string
formatting. Seems silly for constants - formatting is only useful for
outputting, not inputting, in my opinion.
```

1.114 lsfl

```
LISTFILE      Not all assemblers support this

Format:       LISTFILE [']filepath[']

The path is relative to Tandem's PROGDIR:

Tandem places PRT: in a buffer when it begins an assembly, and if it
encounters LISTFILE it replaces it with the above filepath.
```

1.115 cmac

```
CMACRO      Tandem does not support this pseudo op.

A CMACRO is like a MACRO, but can be referenced case blind.

680x0 assembly language consistently follows these rules:

   - labels are case significant
   - other program elements (e.g. opcodes, reg names) are case blind

CMACRO spoils this neat consistency.

If Tandem finds a CMACRO, it treats it as if it were a MACRO. So if you
spell the CMACRO reference as if it were not case blind, it will still
assemble ok.
```

1.116 else

```
ELSE ELSEIF  Tandem does not support these pseudo ops

The use of ELSE and ELSEIF as well as  IFcc...ENDC  makes program structure
too hard to read. This is an example of ill conceived fiddling with assembly
language, threatening compatibility between assemblers, for no good reason.
```

1.117 rept

```
REPEAT     Tandem does not support this pseudo op.

Alternate form:    REPT

Causes Tandem to repeat a section of sc. Causes a fatal error if found. Of
course, MACRO...ENDM should be used. This is a useless extension, and a
threat to compatibility. People can't stop fiddling.
```

1.118 bopt

```
                SUPER BOPT         Tandem does not support these

SUPER    is meant to allow supervisor instructions
BOPT     has a series of built-in options for optimising &c

Tandem has fixed rules for optimising, except for treatment of .L relative
branches. See:

for .L relative branches
                Relative Branching
                for address optimisations
                Actual op Address Format
```

```
                for priveleged instructions, see
                Errs &c
                Therefore, Tandem does not support the above pseudo ops; Tandem's  ↩
                   method
does not cause incompatibility problems between assemblers.
```

1.119 prtx

```
PRINTX DOSCMD       Tandem does not support these

PRINTX prints a string to the Stdout (why not the assembly list?)
DOSCMD executes an AmigaDOS command

These seem to result from confusion about the purpose of an assembler. An
assembler is not a CLI.
```

1.120 incb

```
INCBIN IBYTES           Not all assemblers support these pseudo ops

Format:     INCBIN [']filename[']  [,mxsize]   } Tandem ignores
            IBYTES [']filename[']  [,mxsize]   }   mxsize

INCBIN/IBYTES takes a file, and inserts it in a DS.B, if it will fit. If it
won't fit, an error will occur. A DS.B will be created, equal to the
filesize, and the file will be loaded therein. This is a useful command, but
the trouble is that more often than not the file would be wanted in chip
memory, but the program will be loaded in public mem. So, in practice most
programmers will plan to allocate chip mem and load the file from the
PROGDIR: at run time.


DSBIN                   Tandem does not support this

A few assemblers use this to create an empty block the same size as a file,
but this seems useless, so Tandem does not support it.
```

1.121 incd

```
                INCDIR INCPATH        Not all assemblers recognise these pseudo  ↩
                   ops
                              Use discouraged

When Tandem starts, it first points the CD to its own PROGDIR: (which is
usually Work:Tandem), and then issues this AmigaDOS command:

  ASSIGN >NIL: INC: Includes ADD

thus, the INC: dir (if it didn't already exist) will be created, and pointed
to  Work:Tandem/Includes.
```

```
It is traditional in the Amiga for the INC: dir to be used for include
files. You should save all include files in INC:, giving them the suffix .i

INCDIR or INCPATH assigns a dir to INC:, like this:

Format:    INCDIR [']dirname[']       (or INCPATH for INCDIR)

Action:    assign >nil: INC: dirname

(Tandem uses the dos.library _LVOExecute to do this; it does not check the
validity of the dir you specify, and does not report failure).

The dirname must NOT contain spaces or semicolons, even between quotes, nor
may it contain embedded quotes.

'dirname' will be interpreted by AmigaDOS relative to Tandem's PROGDIR:, so
in general you should put the complete path in dirname. e.g. if you have
includes for the sc you are assembling in WORK2:Projects/Includes, you'd
put:

      INCPATH 'WORK2:Projects/Includes'

before the first INCLUDE. But note that Tandem loads IncAll.i from INC: when
it boots up, and leaves it in the Includes buffer even if you flush the
buffer. But if you re-size the Includes buffer, Tandem will then try to load
IncAll.i from your new INC:, where it presumably won't find it, which will
disable Tandem's INCLUDE 'IncAll.i' feature.

For more on INCLUDE, see
                INCLUDE
                .

In general, it is better to use a full path for includes kept in an unusual
place, rather than use INCDIR/INCPATH. So, suppose you have some includes
in work:MyIncs instead of Tandem/Includes

bad:      INCDIR "work:MyIncs"              ;re-assign INC:
          INCLUDE "Bells.i"
          INCDIR "work:Tandem/Includes"     ;restore INC:

good:     INCLUDE "work:MyIncs/Bells.i"     ;leave INC: alone

The 2nd alternative avoids using any extension pseudo ops, which threaten
compatibility.
```

1.122 pure

```
FILECOM PURE       Tandem does not support these

Some assemblers use these to cause a comment to be attached to the mc file,
or to set its pure bit. Tandem does not support these. You would have to use
the CLI "by hand" after saving the mc.
```

1.123 dstr

```
DSTRING    Tandem does not support this

Some assemblers use this to insert the date of assembly into the mc, at the
cost of compatibility. This is ill conceived, since the mc would fail to
be one-to-one with the sc, a lazy and poor way of documenting.
```

1.124 tsym

```
                Tandem Symbols

Tandem has 4 symbol tables:

1. The symbols from the most recent assembly (null until you do an assembly)
2. The symbols loaded from the file  Tandem/Support/incall.consts
3. The symbols loaded from the file  Tandem/Support/ssxref.consts
4. The symbols loaded from the file  Tandem/Support/custom.consts (if any)

ssxref.consts symbols

ssxref.consts contains all the symbols the Amiga OS3.1 FD files, i.e. all
the _LVO's for calling the ROM Kernal libraries. It also contains
_AbsExecBase.

Also, Tandem has its own internal library, tandem.library, which you can use
if you want to (see
                Front.i
                 for details); all the _LVO's for
tandem.library are also in ssxref.consts.

incall.consts

incall.consts contains all the symbols defined in the OS3.1 .i files, except
for the MACRO definitions, which are in Tandem/Includes/IncAll.i

custom.consts

custom.consts is available for you to use for your own set of constants.
e.g. you might wish to write your own library, and save its _LVO's and other
relevent constants.

Normal Assembly

When Tandem begins an assembly, the only symbol table it has is the first
one. It starts off with:

D0-D7,A0-A7,SP,PC and ZPC  -  the register names (in all possible cases)
NARG                       -  initially zero - see
                MACRO
                As your program's assembly proceeds, Tandem will add to its symbol ↩
                    table all
the symbols it finds.
```

Also, if Tandem finds an XREF, it will:

(i) Seek the XREF symbol in the ssxref.consts table. If it finds it, it will get its value from there, and place the symbol in the assembly symbol table, with type absolute.

(ii) If the XREF symbol is not in the ssxref.consts table, it will store your symbol in the assembly symbol table, with type xref. This means that Tandem will be able to save your program as an object file, but not as an mc file. Your program could then only be made into an mc file by a linker, which would have to link your object file with another object file containing an XDEF to match your XREF.

In case (ii), the commonest reason why an XREF is not in the ssxref.consts table is because it is a misspelled _LVO. In that case, You can click

Rel exts
in the Main window, and seek the next external XREF, which will find the culprit (hotkey Ctrl/X). In other words, few users will ever deliberately use XREF's which are not in ssxref.consts, and few users will ever want to make object files to be linked with a linker.

Consider this sc fragment, for an example of Tandem symbols:

```
    XREF _AbsExecBase
    XREF _LVOOpenLibrary,_LVOCloseLibrary
    XREF Fred
    ....
   libvers: EQU 40
    ....

   OpLib:
    MOVE.L _AbsExecBase,A6
    LEA libname,A1
    MOVEQ #libvers,D0
    JSR _LVOOpenLibrary(a6)
    TST.L D0
    RTS
```

The XREF's _AbsExecBase, _LVOpenLibrary, and _LVOCloseLibrary are in ssxref.costs. So, Tandem will store them in the symbol table as absolute constants.

the XREF Fred is not in ssxref.consts, so Tandem will put it in the symbol table as an xref constant, and so you will be only able to store the program as an object file, not as mc.

Tandem will store the symbol libvers as an absolute symbol, and OpLib as a relative symbol.

Assembling with IncAll.i

If your program contains the following line:

```
    INCLUDE 'IncAll.i'
```

then from that line onward, not only are the MACRO's in IncAll.i available

as you would expect, but also Tandem makes all the symbols in incall.consts, ssxref.consts, and custom.consts (if any) available. The ssxref.consts no longer need XREF's to access them. If you do make XREF's to ssxref.consts, it is not a problem since Tandem will simply ignore them.

This is a terrific feature of Tandem, which suddenly makes Amiga assembly language much easier and more convenient. No more wondering which of the Amiga OS3.1 includes to include - they are all present, and all instantly assembled! But there are three disadvantages:

1. If you have already defined any symbols before you INCLUDE 'IncAll.i', and they duplicate symbols within IncAll.i, Tandem will not notice the double definition error. e.g. If you do this:

   ```
   Menu: EQU 20
   .....
    INCLUDE 'IncAll.i'
   ```

   Then if you subsequently refer to Menu, the Menu in your program will be referenced, not the Menu in the Amiga OS3.1 .i files. This can lead to puzzling bugs.

   Solution: put INCLUDE 'IncAll.i' before any statements that create symbols. (This is normal practice with includes anyway).

2. You cannot use symbols in the Amiga OS3.1 .i files in your program, or you'll get a double definition error. e.g. if you have:

   ```
    INCLUDE 'IncAll.i'
    ....
   Menu:
   ```

   then you'll get an assembly error. Since there are many thousand symbols in incall.consts, that's a lot of symbols you can't use.

   This is only a small problem, since any mistakes you make will show up as assembly errors and are easily fixed. Actually I think it's an advantage, since it effectively "reserves" the Amiga OS symbols.

3. Other assemblers do not support IncAll.i. So, to make your programs portable, you would have to:

   (a) put in all the XREF's anyway, even though you don't need them, after INCLUDE 'IncAll.i'

   (b) put in all the Amiga OS3.1 .i INCLUDE's needed anyway, even though you don't need them.

   If you do the above, and comment out the INCLUDE 'IncAll.i', and your program still assembles free of assembly errors, then your program will be portable to other assemblers (note: not all assemblers are capable of assembling all the Amiga OS3.1 .i files, believe it or not).

   It is easy to find the symbols you need XREF's for, by clicking "View sym" in the Main window, since they all start with an _ (underscore).

Making Your own pre-assembled files - Advanced Usage

1. To make custom.consts

   Assemble the sc which creates all the consts you need. If you want _LVO's in your program, assemble them as EQU's, e.g.

   ```
   _LVOMyProg1: EQU -30
   _LVOMyProg2: EQU -36   etc.
   ```

   Then, select "Save symbol Table" in the Main window menu. Tandem will save all absolute symbols (except NARG, and synthetic labels made with _nnn at their start, from \@) in Tandem/Support/custom.consts

2. To make ssxref.consts.

   Suppose Amiga brings out OS3.5. There would then presumably be a new Developer's CD. In that, you would look for the FD files. Having found the FD files:

   (a) use C:join to consolidate the FD files into 1 big FD file, called e.g. Tandem/Support/FD3.5 - and to this, also add Tandem/Support/Tandem.FD - and also, if you have any other libraries you have written, you can make FD files in the same format as Tandem.FD to also include.

   (b) then, load the sc file Tandem/Support/ssxref_make.asm
   Check that the input and output files are what you want, e.g.:

   ```
   infile:  Support/FD3.5
   outfile: Support/ssxref.asm
   ```

   assemble and run ssxref_make.asm, and it will create the outfile (i.e. ssxref.asm in the above case). Then, load ssxref.asm, and append RTS on the end, to make it assemble something. Assemble it, and click "Save Symbol Table" from the Main window, and select ssxref.consts

3. To make incall.consts

   This is a big job. Suppose Amiga brings out OS3.5. There would presumably be a new Developer's CD. In that, you would look for the Include .i files.

   (a) make an inspection of the Exec/library files, seeing which ones to put in what order, beginning with Exec/types.i, so that nothing tries to INCLUDE anything not yet appended. Go on to other files, appending things in order. You will find the following problems:

   - a few files (e.g. Intuition/Intuition.i and Intuition/iobsolete.i) try to INCLUDE each other. You would then need to comment out the first of these so that it doesn't try to include the other.
   - a couple of files (e.g. devices/cd.i) have omitted the IFD..ENDC around the INCLUDE's, so you need to comment these out.
   - 2 rubbish files, viz:
     Exec/Exec.i and Exec/Exec_lib.i
     cannot be included.

   (b) append any of your own supporting files fo your own libraries, if you

```
       want. However it might be better to put those in custom.consts.

   (c) append Tandem/Support/Tanlib.i

   This will make a large combined file: OS3.1 took about 650000 bytes,
   which required an SC buffer of $B00000, labels pointers $20000,
   labels ascii $40000, and asmlines $100000 (other buffers normal).

   (d) finally, append RTS so there is something to assemble. After
       successful assembly, click "View sym" to see that everything looks
       ok, and if so, select "Save Symbol Table" from the Main window menu,
       and select incall.consts.

   (e) now, seek out all MACRO definitions (check that there are no
       defns using lower case e.g. Macro or macro), and if there are make
       them MACRO first, to be easier. Use cut & paste to remove all lines
       from between the MACRO definitions, thus leaving sc to consist of
       nothing but uncommented MACRO definitions with no white space.
       Save this file as IncAll.i in Tandem/Includes/IncAll.i
```

1.125 texp

```
                Tandem Expressions

The expressions that occur in sc can be of several different types, i.e.:

1. Absolute expressions
   These have a fixed numerical value

2. Relative expressions
   These have values relative to where a program is loaded in memory

3. XREF's
   These have a value which is undefined until a linker gives them a value
   n.b. Tandem resolves _LVO XREF's in OS3.1 ROM Kernal Amiga libraries,
   into absolute values, so for example _LVOAllocVec is an absolute operand,
   not an XREF operand. See
                XREF
                 for details.

4. EQUR
   These are equal to register names, from D0-D7 or A0-A7

5. REG
   These are equal to a list of register names, as in MOVEM instructions

An expression of types 3-5 above consists only of 1 element, i.e. the label
which has the required value. e.g. if    Fred EQUR A4,  then Fred is an
expression, but  fred+4  is not, since EQUR REG and XREF type symbols cannot
be operated on in expressions by operators like + or - but must stand alone.
A relative operand can occur in a limited range of contexts, but most often
will occur alone. And an absolute operand may occur in just about any
context.

Here are the rules for expressions:
```

```
1. an XREF operand must occur alone in its expression

2. an EQUR operand must occur alone in its expression

3. a REG operand must occur alone in its expression

4. a relative operand:

     (i)   cannot have unary operators

     (ii)  may occur in the context:
           relative - relative
           when the result is absolute

     (iii) may occur in the context
           relative + absolute
           relative - absolute
           when the result is relative

     (iv)  may occur in parenthesis

   no other operations are possible for relative operands

5. expressions (subject to rules 1-4) consist of:

     (i)   at least 1 operand

     (ii)  each operand might have 0 or more leading unary operators

     (iii) each pair of operands must be separated by exactly 1 binary
           operator

     (iv)  in evaluating an expression, parenthesis takes precedence over
           unary operators, which take precedence over binary operators.

6. operands can be:

     (i)   decimal numbers

     (ii)  hexadecimal (hex) numbers, denoted by a leading $

     (iii) binary numbers, denoted by a leading %

     (iv)  strings, delimited with ' or ". Within '..', successive '
           characters are interpreted as a single embedded '. Likewise ".

     (iv)  labels, as follows:

           (a) absolute type values, by using them in EQU or SET statements
           (b) XREF type values, by using them in XREF statements
           (c) EQUR type values, by using them in EQUR statements
           (d) REG type values, by using them in REG statements
           (e) labels of MACRO statements cannot be used as operands
           (f) all other labels have relative type values

     (v)   *  which means the relative address of the current line
```

(vi) expressions in parenthesis

7. unary operators can be

```
+    no effect
-    negation
~    complementation
```

they are of equal precedence, evaluated right to left.

8. binary operators can be (in increasing order of precedence)

```
+    plus
-    minus
*    multiplied by
/    divided by (2nd operand<>0)
!    or
&    and
^    eor
<<   left shift
>>   right shift
```

where of equal precendence, they are evaluated left to right

9. Tandem uses 32 bit arithmetic to evaluate expressions

10. If a hex operand has 2 or 4 digits and no extension, its ms digit is not sign extended. (this is consistent with other assemblers).

```
e.g. $FE    is taken to be $000000FE
     $7E    is taken to be $0000007E
     $8100  is taken to be $00008100
     $7C12  is taken to be $00007C12
```

It is hard to specify a satisfactory rule here; either sign extending or not sign extending can both lead to counter-intuitive results. So, e.g. if you are working on a .W length expression, use 4 digits for your hex -ve constants. And 8 for .L hex -ve constants.

Tandem treats all decimal constants as .L

11. Checking the value range of an expression (if .B or .W is required)

(a) relative expressions are always .L

(b) Tandem uses the following operation to test if D0 (abs) is .B

```
move.b d0,d1
ext.w d1
ext.l d1
cmp.l d0,d1
```

if the result is NE, then D0 is out of range

(c) Tandem uses the following operation to test if D0 (abs) is .W

```
          move.w d0,d1
          ext.l d1
          cmp.l d0,d1

        if the result is NE, then D0 is out of range
```

1.126 error

```
                Error Codes

Er 1 (unused)

Er 2 Attempt to double-define a label
     Only a SET may be redefined.  e.g.   fred equ 2 ... fred set 3  is bad

Er 3 XDEF/XREF s/be non-local label
     e.g. XREF .fred   is not allowed since  .fred  is a local label

Er 4 .B/.W/.L in bad context
     e.g. MOVEA.B D0,A0    is bad since MOVEA requires .W or .L extension

Er 5 Scale/Register in bad context
     e.g. TST A0*2   is bad since scaling not allowed in this addr mode

Er 6 68040 and 68060 not supported
     Tandem can only assemble up to 68030 (and MMU,68881/2)

Er 7 (unused)

Er 8 Unrecognised address mode
     e.g.  movea.l a0<a1      has a garbled address field

Er 9 Must be rel or abs expression
     e.g. moveq #d0,d1   is bad, since an expression expected after #

E 10 Missing operand/address
     e.g. no expression where one s/be or no operand after a binary operator

E 11 Bad address syntax
     A catch-all for garbled addresses, especially in pseudo ops

E 12 $/% must have hex/binary chrs
     e.g. tst $     is bad since hex characters expected after $

E 13 Undefined label in expression
     perhaps a label was misspelled?

E 14 OS var w'out INCLUDE 'IncAll.i'
                INCLUDE
                     e.g. if you reference, say, RP_JAM2, but no INCLUDE for it.

E 15 MACRO label in an expression
     a MACRO label cannot appear in an address

E 16 Can't reference an EQUR/REG lab
```

```
     EQUR/REG labels must not occur with operators like + or - or ()

E 17 (unused)

E 18 Bad ([]) nesting

E 19 Can only do - or ~ on abs ops
                Expressions
                     e.g.  -fred  is not allowed if fred is relative

E 20 Inappropriate math operation

E 21 rel bin rel can only be -
                Expressions
                     e.g.  fred-bill  (both rel) is ok, but not  fred+bill

E 22 rel bin abs can only be + or -
                Expressions
                     e.g.  fred+4, fred-4 ok (fred rel), but not fred/4

E 23 abs rel bin not allowed
                Expressions
                     e.g.  4+fred (fred rel) not allowed: use  fred+4

E 24 / 2nd operand s/be <>0
                Expressions
                     e.g.  4/0  not allowed

E 25 Value out of range of .B/.W/.L
     e.g. MOVEQ #500,D0

E 26 Extra chars after end of exprssn
     maybe ; omitted from comment

E 27 Chr after \ must be @ or 0-9
                MACRO
                     i.e. bad MACRO parameter syntax

E 28 Bad value in expression
     e.g.  PLEN 0  is bad, since this is an illegal value for PLEN.

E 29 Bad <...> usage in MACRO refrnce
                MACRO
                E 30 (Fatal) Overflow - Labels ASCII
                Mem size
                E 31 (Fatal) Overflow - Labels pnters
                Mem size
                E 32 (Fatal) Overflow - Includes
                Mem size
                E 33 (unused)

E 34 (Fatal) Overflow - Object code
                Mem size
                E 35 (Fatal) Overflow - Assmbly Lines
                Mem size
                E 36 (Fatal) Overflow - Relocations
                Mem size
```

```
                E 37 (Fatal) Overflow - XDEFs/XREFs
                Mem size
                E 38 (Fatal) ENDC without prior IFcc
                IFcc
                E 39 (Fatal) Can't find/load INCLUDE
                INCLUDE
                E 40 (Fatal) More than 50 errors
     Tandem stops assembling if >50 errors occur

E 41 (Fatal) IFcc..ENDC nesting > 10
                IFcc
                E 42 (Fatal) MACRO ref nesting > 10
     Are 2 MACRO's referencing each other?

E 43 (Fatal) INCLUDE nesting > 10
     Are 2 INCLUDE files INCLUDE'ing each other?

E 44 (Fatal) MACRO without ENDM
     Bad MEXIT usage?  n.b. MACRO definitions cannot be nested

E 45 (Fatal) IFcc without ENDC
     Be careful in using IFcc..ENDC - the unmatched IFcc can be hard to find

E 46 (Fatal) FAIL found
                FAIL
                E 47 (Fatal) Overflow - Pass 2 exprns
                Mem size
                E 48 (Fatal) Pointer top left of scrn
     You can abort an assembly by moving the pointer to screen top left

E 49 (Fatal) mc has become unaligned
     Usually caused by a missing  DS.W 0  after  DS.B/DC.B

E 50 Bad syntax: nnn without $
     Caused by spurious numbers at start of line

E 51 Bad character at start of line
                Format Rules
                E 52 Bad character at end of field
     Space omitted between opcode & address?

E 53 Cannot identify opcode
     Opcode misspelled?

E 54 (unused)

E 55 (unused)

E 56 ENDM found without prior MACRO

E 57 MEXIT not between MACRO...ENDM

E 58 This statement must be labelled
     SET/EQU/EQUR/REG/MACRO must be labelled

E 59 Pseudo op has forward reference
     The only pseudo that can have a forward reference is DC
```

```
E 60 (unused)

E 61 EQU/SET must be an abs or rel
     e.g. you cannot have EQU followed by an EQUR label

E 62 Wrong addr mode for this opcode
     e.g.  TST.L A0  is not allowed

E 63 (unused)

E 64 Expression value out of range
     e.g.  MOVEQ #$1234,D0    this opcode requires -128 to +127

E 65 (unused)

E 66 #exp s/be type abs, value 1-8
     e.g.  ASL #9,(A4)        s/be 1-8

E 67 Bcc,BSR,DBcc exp s/be relative
     it is best to always make the address a label (don't use *)

E 68 Bitfield {offset:width} missing
     BFCHG &c require a bitfield, e.g.  BFCHG D0{3:2}

E 69 Bad bitfld {offset:width} syntax
     See E 68 above

E 70 Bitfield values must be absolute
     See E 68 above

E 71 DBcc address out of range
     A DBcc can only be .W in length. In practice, would always be much less

E 72 .S/.X/.P/.D ext s/be 68881 only
     These are FPU extensions, e.g. FMOVE.S (A0),FP1 is ok

E 73 (unused)

E 74 Ext here should be .X
     Applies to FPU math instructions

E 75 Ext with Dn must be .B .W .L .S
     Applies to FMOVE with the second address Dn, e.g. FMOVE.S FP0,D3 is ok

E 76 {k} missing from FMOVE.P FPn,ea
     FMOVE.P requires a suffix, e.g.  FMOVE.P FP3,(A3){#4}  is ok

E 77 Bad {k} syntax in FMOVE.P
     Did you omit the #?

E 78 Bad reglist syntax
     Tandem does not allow wrap around, e.g. D6-D1 is bad, use D6-D7/D0-D1
     Mixing D & A is permitted, e.g. D0-A6, but D0-D7/A0-A6 is better.

E 79 Bad extension:  .X required
     Applies to FMOVEM of FP registers
```

```
E 80 Bad extension:  .L required
     Applies to FMOVEM of non-FP registers, e.g. FPCR

E 81 Bad FPc:FPs syntax in FSINCOS
     There is special 68881 syntax for FSINCOS

E 82 (unused)

E 83 TRAP must have value 0-15
     e.g. TRAP #15 is ok, not #16

E 84 PFLUSH # value(s) must be 0-7

E 85 Not PMOVEFD with MMUSR

E 86 Expression here must be type abs
     Mainly applies to pseudo ops

E 87 IFD &c label exists, wrong type
     cannot use a MACRO label for IFD/IFND - use IFMACROD/IFMACROND

E 88 Bad IFC/IFNC syntax
     check usage of apostrophes, e.g. IFC \1,'' is ok

E 89 relative expression must be .L
     e.g. TST (fred.w,A0) is bad if fred is relative (68020 syntax)

E 90 EQUR addr s/be D0-D7 or A0-A7
     Tandem cannot have EQUR for CCR,SR or PC

E 91 This pseudo-op is not supported
                Extension Pseudo ops
                E 92 XREF item already exists locally
     An XREF cannot be defined in the program - did you mean XDEF?

E 93 no SECTIONs used - all 1 hunk
                SECTION
                E 94 can't load INCBIN file

E 95 redefinition of incall.i label
                INCLUDE
                     If you use INCLUDE 'IncAll.i', all OS3.1 labels become pre- ↩
                        defined. So
     e.g. you can't then define use Menu as a label since it already exists.

E 96 d of d(PC)/d(PC,Xi) s/be reltive
     e.g. 4(PC) is bad - but fred(PC) is ok, if fred is a relative label.

E 97 bad floating pt constant syntax
     See
                DC
                 for the rules for floating point constants.

E 98 not allowed in immediate mode
                Immediate Mode
```

E 99 (unused)

1.127 mc

Assembly Language and Machine Code

Tandem can assemble 68030 assembly language. However this appendix only lists 68000 machine code format. For higher 680x0, you would need to refer to a book.

68000 Assembly Language

68000 Machine Code
See also:

Rules for Source Code

1.128 alang

Appendix - 68000 Assembly Language

codes for flags: *=affected -=unaffected 0=cleared 1=set ?=undefined
codes for extensions: .x = .L .W .B .y = .L .W .z = .L .S

(Except where noted, Tandem assumes .W if the extension is omitted)
(I recommend you use .W/.B instead of .L/.S
- see
Relative Extensions
)

I suggest you always use 68000 compatible, unless your program requires AGA &/or FPU (and perhaps even then).

Addressing Modes

Dn Data register direct e.g. D4
contents of D register

An Address register direct e.g. A5
contents of A register

(An) Address register indirect e.g. (A2)
memory at address An

(An)+ Address register indirect with post increment e.g. (A7)+
memory at address An, add bytes read to An after reading

-(An) Address register indirect with pre decrement e.g. -(A7)
reduce An by bytes read before reading, memory at new address

```
d(An)  Address register indirect with displacement    e.g. 20(A3)
       memory at address An+d,  $8000<=d<=$7FFF

d(An,Xm.y)  Address register indirect with index      e.g. 10(A4,D6.W)
       X is A or D. Memory at An+Xm.y+d,  $80<=d<=$7F

xxx.y  Absolute short/long address      e.g. $00000004
       memory address xxx  (if .y omitted, Tandem assembles .L)

d(PC)  program counter with displacement     e.g. -20(PC)
       as per d(An), PC replaces An

d(PC,Xm.y)  Program counter with index       e.g. -5(PC,A3.L)
       as per d(An,Xm.y), PC replaces An

#xxx  immediate data                         e.g. #15
       range of values as per opcode extension, or 1-8 as indicated below

SR     status register

CCR    condition code (flag) register


ABCD     Add binary coded decimal

Flags:   X* N? Z* V? C*
Syntax:  ABCD Dn,Dn
         ABCD -(An),-(An)
Action:  Adds X+1st addr to 2nd addr using binary coded decimal; sets X

ADD      Add binary

Flags:   X* N* Z* V* C*
Syntax:  Add.x EA,Dn     EA=Dn An (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
         Add.x Dn,EA     EA=      (An) (An)+ -(An) d(An) d(An,Xm.y)
Action:  Adds 1st addr to 2nd addr

ADDA     Add binary to A register (Tandem allows ADD)

Flags:   none
Syntax:  ADDA.y EA,An    EA=Dn An (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
Action:  as per ADD

ADDI     Add immediate   (Tandem allows ADD)

Flags:   X* N* Z* V* C*
Syntax:  ADDI.x #xxx,EA  EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  as per ADD

ADDQ     Add quickly

Flags:   X* N* Z* V* C*
Syntax:  ADDQ.x #xxx,EA  EA=Dn An (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  as per ADD      xxx is from 1 to 8 built in to opcode for speed
```

```
ADDX     add with extend

Flags:   X* N* Z* V* C*
Syntax:  ADDX.x Dn,Dn
         ADDX.x -(An),-(An)
Action:  adds X+1st addr to 2nd addr

AND      logical and

Flags:   X- N* Z* V0 C0
Syntax:  AND.x EA,Dn     EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
         AND.x Dn,EA     EA=      (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  logical ands 1st addr into 2nd addr

ANDI     and immediate   (Tandem allows AND)

Flags:   X- N* Z* V0 C0
Syntax:  ANDI.x #xxx,EA  EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  as per AND

ASL and ASR      arithmetic shift left/right

Flags:   X* N* Z* V* C*  (V set if high bit changes during the operation)
Syntax:  ASR.x Dn,Dn
         ASR.x #xxx,Dn
         ASR.W EA        EA=      (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  if 2 addresses, shifts 2nd address 1st address times
         if one address, shifts address 1 time (word length)

            +---+                                C and X are the last bit
            | C |<-+                                    shifted out
            +---+  |    +----------+    +---+   vacated bits are zero filled
  Left             +<---| Operand |<---| 0 |
            +---+  |    +----------+    +---+   effect is signed double
            | X |<-+
            +---+

                                        +---+    C and X are the last bit
                                     +->| C |             shifted out
                      +----------+   |  +---+
  Right           +->| Operand |--->+           leftmost bit duplicated
                  |  +----------+   |  +---+        in vacated bits
                  |   |             +->| X |
                  +---+                +---+    effect is signed halve

Bcc     branch conditional

Flags:   none changed
Syntax:  Bcc.z <label>
Action:  Branches depending on the values of the flags (except for X)

         BRA  always  (also written BT)                    1
         BEQ  EQ                                           Z
         BNE  NE                                           ~Z
         BPL  PL                                           S
         BMI  MI                                           ~S
         BVS  VS                                           V
```

```
        BVC  VC                                                ~V
        BCS  CS                                                C
        BCC  CC                                                ~C
        BGE  flags per signed compare >=                       N*V+~N*~V
        BLE  flags per signed compare <=                       Z+N*~V+~N*V
        BLT  flags per signed compare <                        N*~V+~N*V
        BGT  flags per signed compare >                        N*V*~Z+~N*~V*~Z
        BHS  flags per unsigned compare >= (high or same)      ~C
        BHI  flags per unsigned compare >  (high)              ~(C*Z)
        BLS  flags per unsigned compare <= (low or same)       C+Z
        BLO  flags per unsigned compare <  (low)               C

        For Bcc.L the PC can only go forward up to 32766 bytes or
            backward up to 32768 bytes, measured from opcode+2.
        For Bcc.S the PC can only go forward up to 126 bytes or
            backward up to 128 bytes, measured from opcode+2. .S is
            quicker than .L and takes only 1 word of mc. A .S cannot
            jump forward 0 bytes.
        For backward Bcc's, Tandem assembles .S if possible else .L, and
        ignores the extension. Jumps longer than Bcc.L must use JMP

n.b. do not use the .L extension - omit it, and .L is assumed. In 68020+
syntax, .L is written .W, and .L is used for 32 bit branching. See

                Relative Branching
                 for a discussion of this topic. You can
use .B instead of .S

BCHG     Bit change

Flags:   Z set to what the bit was changed to; others unchanged
Syntax:  BCHG Dn,EA      EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
         BCHG #xxx,EA    EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Changes the bit number of EA. If EA is Dn, the bit no. is 0-31. If
         EA is not Dn, the bit no. is 0-7 (i.e. a byte).

BCLR     Bit clear

Flags:   Z set to opposite of bit cleared before clearing; others unchanged
Syntax:  BCLR Dn,EA      EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
         BCLR #xxx,EA    EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Clears the bit number of EA, addressing as per BCHG

BSET     Bit set

Flags:   Z set to opposite of bit set before setting; others unchanged
Syntax:  BSET Dn,EA      EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
         BSET #xxx,EA    EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Sets the bit number of EA, addressing as per BCHG

BSR      Branch to subroutine

Flags:   none
Syntax:  BSR.z label
Action:  Branches to a subroutine at label. BSR.S is almost never used in
         practice; it can branch forward 2 to 126 bytes or back 2 to 128
         bytes from opcode+2. BSR.L (the .L is generally omitted) can brnch
```

```
         forward up to 32766 bytes or back up to 32768 bytes. For longer
         jumps use JSR.

See the note about extensions under Bcc.

BTST     Bit test

Flags:   Z set to opposite of bit tested
Syntax:  BTST Dn,EA      EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,An)
Action:  Tests the bit number of EA, addressing as per BCHG

CHK      Check against bounds

Flags:   X- N* Z? V? C?
Syntax:  CHK EA,Dn       EA=Dn An (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(Pc,Xm.y) #xxx
Action:  Causes an exception if Dn.W<0 or Dn.W>EA. An exception crashes
         the system if you do not have an interrupt server attached.

CLR      Clear

Flags:   X- N0 Z1 V0 C0
Syntax:  CLR.x EA        EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  sets EA to zero

(n.b. MOVEQ #0,D0 is quicker than CLR.L D0)

CMP      Compare

Flags:   X- N* Z* V* C*
Syntax:  CMP.x EA,Dn     EA=Dn An (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
Action:  Leaves Dn unchanged, but sets flags as if it was SUB

CMPA     Compare A register  (Tandem allows CMP)

Flags:   X- N* Z* V* C*
Syntax:  CMPA.y EA,An    EA=Dn An (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,An) #xxx
Action:  Leaves An unchaged, but sets flags as if it was SUBA

CMPI     Compare immediate   (Tandem allows CMP)

Flags:   X- N* Z* V* C*
Syntax:  CMPI.y #xxx,EA  EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Leaves EA unchanged, but sets flags as if it was SUBI

CMPM    Compare memory   (Tandem allows CMP, but I recommend CMPM)

Flags:   X- N* Z* V* C*
Syntax:  CMPM.y (An)+,(An)+
Action:  Leaves memory unchanged, but sets flags as if it were a SUB

DBcc     Decrement, Test and Branch

Flags:   none
```

```
Syntax:  DBcc Dn,label
         cc as per Bcc, except there is an extra condition:
           DBF, usually written DBRA  (c.f. BT which is written BRA)
           F means never, logic 0
         The commonest form by far is DBRA
Action:  If cc is true (for DBRA this is never), the PC falls through
         If cc is false (for DBRA this is always):
            Dn is decremented by 1
            If Dn becomes -1, the PC falls through
            If Dn doesn't become -1, the PC branches to label
         The PC can only go forward up to 32766 bytes, or backward up to
         32768 bytes, measured from opcode+2.

DIVS     Divide signed

Flags:   X- N* Z* V* C0
Syntax:  DIVS EA,Dn      EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
Action:  divides 32 Dn.L by EA.W, quotient in Dn.W, remainder in high order
         word of Dn. Caution: causes exception if EA=0 (crashes system). If
         overflow occurs, no exception, but result,N,Z invalid.

DIVU     Divide unsigned

Flags:   X- N* Z* V0 C0
Syntax:  DIVU EA,Dn      EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
Action:  as per DIVU, but unsiged.

EOR      Exclusive or

Flags:   X- N* Z* V0 C0
Syntax:  EOR.x Dn,EA     EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Uses Dn to exclusive or EA, bit by bit. The bits of Dn effectively
         invert the corresponding bits of EA.

EORI     Exclusive or immediate    (Tandem allows EOR)

Flags:   X- N* Z* V0 C0
Syntax:  EORI.x #xxx,EA  EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Uses #xxx to exclusive or EA, bit by bit.

EORI to CCR   Exclusive or immediate to CCR    (Tandem allows EOR)

Flags:   See Action
Syntax:  EORI #xxx,CCR
Action:  bits of #xxx switch flags, as follows:
         bit 0 of #xxx if 1 switches N
         bit 1 of #xxx if 1 switches V
         bit 2 of #xxx if 1 switches Z
         bit 3 of #xxx if 1 switches N
         bit 4 of #xxx if 1 switches X

EORI to SR    Exclusive or immediate to SR     (Tandem allows EOR)

Syntax:  EORI #xxx,SR
```

```
Tandem will assemble this, but it can only be used when the 68000 is in
supervisor mode, so it will crash the system in normal usage.

EXG      Exchange registers

Flags:   none
Syntax:  EXG Rn,Rn
Action:  Swaps any 2 registers among D0-D7 and A0-A7

EXT      Extend

Flags:   X- N* Z* V0 C0
Syntax:  EXT.y Dn
Action:  If .W extends Dn.B to the same signed value in Dn.W
         If .L extends Dn.W to the same signed value in Dn.L

ILLEGAL

Flags:   none
Syntax:  ILLEGAL
Action:  Causes an exception. i.e. crashes the system.

JMP      Jump

Flags:   none
Syntax:  JMP EA          EA=      (An) d(An)              d(An,Xm.y) #xxx
                             d(PC) d(PC,Xm.y)
Action:  as BRA, but EA can be anywhere in memory.

JSR      Jump to subroutine

Flags:   none
Syntax:  JSR EA          EA=      (An) d(An)              d(An,Xm.y) #xxx
                             d(PC) d(PC,Xm.y)
Action:  as BSR but EA can be anywhere in memory. JSR d(A6) is used to make
         Amiga ROM calls.

LEA      Load effective address

Flags:   none
Syntax:  LEA EA,An       EA=      (An) d(An)              d(An,Xm.y) xxx.y
                             d(PC) d(PC,Xm.y)
Action:  Calculate an address into An. e.g. LEA label,A0 is eqivalent to
         MOVEA.L #label,A0 (but quicker). With the modes, whatever the
         adress is, not the contents of the address, gets put into An. For
         e.g. LEA 10(A4),A2 puts A4+10 into A2, since 10(A4) looks at the
         address A4+10.

LINK     Link and allocate

Flags:   none
Syntax:  LINK An,#xxx
Action:  This pushes a data area onto the stack:
         1. First, the old An is pushed to the stack
         2. Then, SP is placed in An
         3. Then, #xxx (usually negative) is added to SP
         The data area is now #xxx in length; its top is 4 bytes below
```

```
         An. (See UNLK for the reversal of LINK). Also, if the LINK is
         at the start of a subroutine, anything pushed by the caller before
         calling starts from 8 bytes above An.

MOVE     Move Data

Flags:   X- N* Z* V0 C0
Syntax:  MOVE.x EA,EA
                     1st EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
                     2nd EA=Dn    (An) d(An) (An)+ -(An) d(An,Xm.y) xxx.y
Action:  Moves data from one place to another

MOVE to CCR

Flags:   All affected
Syntax:  MOVE.W EA,CCR   EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
Action:  Moves data to CCR, to set flags registers as per EORI (q.v.)

MOVE from SR

Flags:   none
Syntax:  MOVE.W SR,EA    EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Moves SR (including CCR) to EA. In processors higher than a 68000
         (e.g. the 68020) this will cause an exception, so it should only be
         used in supervisor mode.

MOVE to SR

Flags:   all affected
Syntax:  MOVE.W EA,SR    EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
Action:  move.w EA to SR, including the CCR. This can only be done in
         supervisor mode.

MOVE USP

Flags:   none
Syntax:  MOVE.L USP,An
         MOVE.L An,USP
Action:  This is used in supervisor mode to change the USP, which is not in
         A7 at the time.

MOVEA    Move to A register  (Tandem allows MOVE)

Flags:   none
Syntax:  MOVEA.y EA,An   EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y

MOVEM    Move multiple       (Tandem does NOT allow MOVE)

Flags:   none
Syntax:  MOVEM.y EA,list EA=      (An) d(An) (An)+ -(An) d(An,Xm.y) xxx.y
                           d(PC) d(PC,Xm.y)
         MOVEM.y list,EA EA=      (An) d(An) (An)+ -(An) d(An,Xm.y) xxx.y
Action:  moves words/longwrds starting from the EA, to or from each register
         in the list until the list is exhausted. Each register can only be
```

```
          specified once, and regardless of the order, the registers are
          sent in the following order:

          data regs low to high, then address orders low to high

          When they are popped back, the receiving order is the reverse of
          the sending order.

          If you specify a .W extension, a word is read from each sender,
          then it is longword extended, and a longword is sent.

          list syntax consists of registers, and/or register ranges
          separated by hyphens, separated by slashes. e.g.:

            D1/A1/D3-D7/A3-A4

          Normally, the receiving EA is -(A7), and the sending EA is (A7)+

MOVEP     Move to/from peripheral

Flags:    none
Syntax:   MOVEP.y Dn,d(An)
          MOVEP.y d(An),Dn
Action:   moves bytes one at a time to even addresses. This is not applicble
          to the Amiga's way of doing things.

MOVEQ   Move quick (you must use the Q - Tandem does not optimise)

Flags:    X- Z* N* V0 C0
Syntax:   MOVEQ #xxx,Dn
Action:   moves a longword from FFFFFF80 to 0000007F to Dn. This is quicker
          than MOVE.L, and MOVEQ #0,Dn is even quicker than CLR.L Dn.

MULS      Multiply signed

Flags:    X- Z* N* V0 C0
Syntax:   MULS EA,Dn       EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                              d(PC) d(PC,Xm.y) #xxx
Action:   signed multiply EA.W by Dn.W, result in Dn.L

MULU      Multiply unsigned

Flags:    X- Z* N* V0 C0
Syntax:   MULU EA,Dn       EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                              d(PC) d(PC,Xm.y) #xxx
Action:   unsigned multiply EA.W by Dn.W, result in Dn.L

NBCD      Negate BCD with extend

Flags:    X* Z* N? V? C*
Syntax:   NBCD EA          EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:   Uses X and EA as input, and negates EA using BCD arithmetic

NEG       Negate

Flags:    X* Z* N* V* C*
Syntax:   NEG EA           EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
```

```
Action:  Negates EA

NEGX     Negate with extend

Flags:   X* Z* N* V* C*
Syntax:  NEGX EA         EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Negates EA, including the X

NOP      No operation

Flags:   none
Syntax:  NOP
Action:  does nothing

NOT      logical not

Flags:   X- Z* N* V0 C0
Syntax:  NOT.x EA        EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  inverts all bits of the EA

OR       logical or

Flags:   X- Z* N* V0 C0
Syntax:  OR.x EA,Dn      EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
         OR.x Dn,EA      EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  or's the 1st address into the receiver, bit by bit. Wherever eithr
         sender of receiver (or both) have a 1, the receiver gets a 1.

ORI      logical or immediate   (Tandem allows OR)

Flags:   X- Z* N* V0 C0
Syntax:  ORI.x #xxx,EA   EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y

ORI to CCR     ORI to SR

does OR to CCR or SR, in same way as EORI to CCR/SR, which see.

PEA      Push effective address to stack

Flags:   none
Syntax:  PEA EA          EA=      (An)             d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,An)
Action:  as LEA, but pushes the address to the stack. The fastest way to
         put pointers into the stack.

RESET    reset

Flags:   none
Syntax:  RESET
Action:  theroetically, causes all external devices to be reset. This must
         not be used in the Amiga. Can be executed only in supervisor mode.

ROL   ROR    Rotate  Left/Right

Flags:   X- N* Z* V0 C*
Syntax:  ROR.x Dn,Dn
```

```
         ROR.x #xxx,Dn
         ROR.W EA        EA=      (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  rotates all the bits around the register. What falls out the end &
         gets pushed in the other side, and the last bit to fall out also
         gets pushed into C. A Register can be rotated 1-8 if #xxx, or any
         amount if sender=Dn. An EA is rotated 1 step.

         +-----------------+                          +-----------------+
         |                 |                          |                 |
ROR      |   +---------+   |   +---+     ROL  +---+   |   +---------+   |
         +-->| Operand |-->+-->| C |          | C |<--+<--| Operand |<--+
             +---------+       +---+          +---+       +---------+

ROXL     ROXR     Rotate with X  Left/Right

Flags:   X* Z* N* V0 C*
Syntax:  ROXR.x Dn,Dn        (ROXL is the same)
         ROXR.x #xxx,Dn
         ROXR.x EA       EA=      (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  As ROL/ROR, but includes X in the rotation.

         +-----------------------+             +-----------------------+
         |                       |             |                       |
ROXR     |   +---------+       +---+    ROXL  +---+       +---------+   |
         +-->| Operand |-->+-->| X |          | X |<--+<--| Operand |<--+
             +---------+   |   +---+          +---+   |   +---------+
                           |                          |
                           |   +---+          +---+   |
                           +-->| C |          | C |<--+
                               +---+          +---+

RTE      Return from Exception

Flags:   all affected
Syntax:  RTE
Action:  pops the SR (including the CCR) and then the PC from the stack. If
         user state is thereby set in SR, it will return to user state. It
         can only be called from supervisor state. If returning from an
         exception, the other registers would have to be popped first.

RTR      Return & restore CCR

Flags:   all affected
Syntax:  RTR
Action:  pops the CCR, but not the SR, from the stack, and then the PC.
         Unfortunately there is no BTR to do the reverse, so it isn't much
         use.

RTS      Return from subroutine

Flags:   none
Syntax:  RTS
Action:  returns from a subroutine. Pops the PC from the stack.

SBCD     Subtract DCB with extend

Flags:   X* N? Z* V? C*
```

```
Syntax:  SBCD Dn,Dn
         SBCD -(An),-(An)
Action:  Subtracts the first address, and X, from the second.

Scc      Set according to conditions

Flags:   none
Syntax:  Scc EA          EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Sets EA.b to all 1's or all 0', as the cc is true or false. The
         cc's are as under Bcc, and also F (false) which always fills EA
         with 0's.

STOP     Stop

Flags:   All affected
Syntax:  STOP #xxx
Action:  puts #xxx into the SR (including CCR) and waits for a certain
         hardware event. STOP can only be exercised from supervisor mode.

SUB      Subtract

Flags:   X* Z* N* V* C*
Syntax:  SUB.x EA,Dn     EA=Dn An (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
         SUB.x Dn,EA     EA=      (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Subtracts 1st address from 2nd address, puts result in 2nd address

SUBA     Subtract from A register  (Tandem allows SUB)

Flags:   none
Syntax:  SUB.y EA,An     EA=Dn An (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
                            d(PC) d(PC,Xm.y) #xxx
Action:  Subtracts EA from An, result in An

SUBI     Subtract immediate    (Tandem allows SUB)

Flags:   X* Z* N* V* C*
Syntax:  SUBI.x #xxx,EA  EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Subtracts #xxx from EA, result in EA

SUBQ    Subtract quick    (must have Q - Tandem does not optimise)

Flags:   X* Z* N* V* C*
Syntax:  SUBQ.x #xxx,EA  EA=DN An (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  Subtracts #xx, where xxx is 1 to 8, from EA. Quicker than SUBI

SUBX     Subtract Extend     (X must be appended)

Flags:   X* Z* N* V* C*
Syntax:  SUBX.x Dn,Dn
         SUBX.x -(An),-(An)
Action:  Subtracts 1st addr and X from 2nd addr, result in 2nd addr

SWAP     Swap halves of a register

Flags:   X- Z* N* V0 C0
Syntax:  SWAP Dn
```

```
Action:  swaps the low order and high order words of a data register.

TAS      Test and set

Flags:   X- N* Z* V0 C0
Syntax:  TAS EA          EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  reads EA.b and sets Z and N for it. Then, sets bit 7 and writes it
         back to EA. In fact, the Amiga hardware may not allow this
         opcode to work properly, so it should never be used.

TRAP

Flags:   none
Syntax:  TRAP #xxx
Action:  Causes an exception to trap vector #xxx

TRAPV

Flags:   none
Syntax:  TRAPV
Action:  Causes an exception if VS set.

TST      Test

Flags:   X- N* Z* V0 C0
Syntax:  TST.x EA        EA=Dn    (An) (An)+ -(An) d(An) d(An,Xm.y) xxx.y
Action:  sets the flags as if EA was moved, but leaves memory unchanged.

UNLK     Unlink

Flags:   none
Syntax:  UNLK An
Action:  Reverses the LINK (q.v.) opcode. Moves An to SP, pops to An from
         SP. This discards memory created by LINK, ready to RTS.
```

1.129 mcz

```
Machine Codes    (codes at end)

bit:     F  E  D  C  B  A  9  8  7  6  5  4  3  2  1  0

ORI      0  0  0  0  0  0  0  0  size  EA-mode  EA-reg-
ANDI     0  0  0  0  0  0  1  0  size  EA-mode  EA-reg-
SUBI     0  0  0  0  0  1  0  0  size  EA-mode  EA-reg-
ADDI     0  0  0  0  0  1  1  0  size  EA-mode  EA-reg-
EORI     0  0  0  0  1  0  1  0  size  EA-mode  EA-reg-
CMPI     0  0  0  0  1  1  0  1  size  EA-mode  EA-reg-
BTST     0  0  0  0  -D-reg-  1  0  0  EA-mode  EA-reg-
BTST     0  0  0  0  1  0  0  0  0  0  EA-mode  EA-reg-
MOVEP    0  0  0  0  -D-reg-  1  x  S  0  0  1  -A-reg-
MOVE.B   0  0  0  1  EA-reg-  EA-mode  EA-mode  EA-reg-
MOVE.L   0  0  1  0  EA-reg-  EA-mode  EA-mode  EA-reg-
MOVE.W   0  0  1  1  EA-reg-  EA-mode  EA-mode  EA-reg-
NEGX     0  1  0  0  0  0  0  0  size  EA-mode  EA-reg-
MOVE<SR  0  1  0  0  0  0  0  0  1  1  EA-mode  EA-reg-
```

```
CLR      0  1  0  0  0  0  1  0  size  EA-mode  EA-reg-
NEG      0  1  0  0  0  1  0  0  size  EA-mode  EA-reg-
MOVECCR  0  1  0  0  0  1  0  0  1  1  EA-mode  EA-reg-
NOT      0  1  0  0  0  1  1  0  size  EA-mode  EA-reg-
MOVE>SR  0  1  0  0  0  1  1  0  1  1  EA-mode  EA-reg-
NBCD     0  1  0  0  1  0  0  0  0  0  EA-mode  EA-reg-
SWAP     0  1  0  0  1  0  0  0  0  1  0  0  0  -D-reg-
PEA      0  1  0  0  1  0  0  0  0  1  EA-mode  EA-reg-
EXT      0  1  0  0  1  0  0  0  1  S  0  0  0  -D-reg-
MOVEM    0  1  0  0  1  A  0  0  1  S  EA-mode  EA-reg-
TST      0  1  0  0  1  0  1  0  size  EA-mode  EA-reg-
TAS      0  1  0  0  1  0  1  0  1  1  EA-mode  EA-reg-
ILLEGAL  0  1  0  0  1  0  1  0  1  1  1  1  1  1  0  0
TRAP     0  1  0  0  1  1  1  0  0  1  0  0  --vector--
LINK     0  1  0  0  1  1  1  0  0  1  0  1  0  -A-reg-
UNLK     0  1  0  0  1  1  1  0  0  1  0  1  1  -A-reg-
MOVEUSP  0  1  0  0  1  1  1  0  0  1  1  0  T  -A-reg-
RESET    0  1  0  0  1  1  1  0  0  1  1  1  0  0  0  0
NOP      0  1  0  0  1  1  1  0  0  1  1  1  0  0  0  1
STOP     0  1  0  0  1  1  1  0  0  1  1  1  0  0  1  0
RTE      0  1  0  0  1  1  1  0  0  1  1  1  0  0  1  1
RTS      0  1  0  0  1  1  1  0  0  1  1  1  0  1  0  1
TRAPV    0  1  0  0  1  1  1  0  0  1  1  1  0  1  1  0
RTR      0  1  0  0  1  1  1  0  0  1  1  1  0  1  1  1
JSR      0  1  0  0  1  1  1  0  1  0  EA-mode  EA-reg-
JMP      0  1  0  0  1  1  1  0  1  1  EA-mode  EA-reg-
CHK      0  1  0  0  -D-reg-  1  1  0  EA-mode  EA-reg-
LEA      0  1  0  0  -A-reg-  1  1  1  EA-mode  EA-reg-
ADDQ     0  1  0  1  -qdata-  0  size  EA-mode  EA-reg-
SUBQ     0  1  0  1  -qdata-  1  size  EA-mode  EA-reg-
Scc      0  1  0  1  condition-  1  1  EA-mode  EA-reg-
DBcc     0  1  0  1  condition-  1  1  0  0  1  -D-reg-
Bcc      0  1  1  0  condition-  --8-bit-displacement--
BSR      0  1  1  0  0  0  0  1  --8-bit-displacement--
MOVEQ    0  1  1  1  -D-reg-  0  ---8-bit-quick-data---
DIVS     1  0  0  0  -D-reg-  1  1  1  EA-mode  EA-reg-
DIVU     1  0  0  0  -D-reg-  0  1  1  EA-mode  EA-reg-
OR       1  0  0  0  -D-reg-  B  size  EA-mode  EA-reg-
SBCD     1  0  0  0  --reg--  1  0  0  0  0  C  --reg--
SUB      1  0  0  1  -D-reg-  B  size  EA-mode  EA-reg-
SUBA     1  0  0  1  -A-reg-  S  1  1  EA-mode  EA-reg-
SUBX     1  0  0  1  --reg--  1  size  0  0  C  --reg--
CMP      1  0  1  1  -D-reg-  0  size  EA-mode  EA-reg-
CMPA     1  0  1  1  -A-reg-  S  1  1  EA-mode  EA-reg-
CMPM     1  0  1  1  -A-reg-  1  size  0  0  1  -A-reg-
EOR      1  0  1  1  -D-reg-  1  size  EA-mode  EA-reg-
AND      1  1  0  0  -D-reg-  B  size  EA-mode  EA-reg-
EXG      1  1  0  0  --reg--  1  --exg-mode---  --reg--
MULS     1  1  0  0  -D-reg-  1  1  1  EA-mode  EA-reg-
MULU     1  1  0  0  -D-reg-  0  1  1  EA-mode  EA-reg-
ABCD     1  1  0  0  --reg--  1  0  0  0  0  C  --reg--
ADD      1  1  0  1  -D-reg-  B  size  EA-mode  EA-reg-
ADDA     1  1  0  1  -A-reg-  S  1  1  EA-mode  EA-reg-
ADDX     1  1  0  1  --reg--  1  size  0  0  C  --reg--
ASR      1  1  1  0  --cnt--  0  size  R  0  0  -D-reg-
ASR      1  1  1  0  0  0  0  0  1  1  EA-mode  EA-reg-
ASL      1  1  1  0  --cnt--  1  size  R  0  0  -D-reg-
```

```
ASL     1  1  1  0  0  0  0  1  1  1  EA-mode  EA-reg-
LSR     1  1  1  0  --cnt--  0  size  R  0  1  -D-reg-
LSR     1  1  1  0  0  0  1  0  1  1  EA-mode  EA-reg-
LSL     1  1  1  0  --cnt--  1  size  R  0  1  -D-reg-
LSL     1  1  1  0  0  0  1  1  1  1  EA-mode  EA-reg-
ROXR    1  1  1  0  --cnt--  0  size  R  1  0  -D-reg-
ROXR    1  1  1  0  0  1  0  0  1  1  EA-mode  EA-reg-
ROXL    1  1  1  0  --cnt--  1  size  R  1  0  -D-reg-
ROXL    1  1  1  0  0  1  0  1  1  1  EA-mode  EA-reg-
ROR     1  1  1  0  --cnt--  0  size  R  1  1  -D-reg-
ROR     1  1  1  0  0  1  1  0  1  1  EA-mode  EA-reg-
ROL     1  1  1  0  --cnt--  1  size  R  1  1  -D-reg-
ROL     1  1  1  0  0  1  1  1  1  1  EA-mode  -D-reg-

codes:

A  0=reg>mem 1mem>reg
B  0result>Dreg 1result>EA
C  0Dregs  1predecEA
R  0cnt=immed  1cnt=Dreg
S  0=.W  1=.L
T  0reg>USP  1USP>reg
X  0mem>Dreg 1Dreg>mem
--cnt--  1 to 8 (000=8)
size  00byte 01word 10longword
--vector-- 0 to 15
-qdata- 1 to 8 (000=8)
-8-bit-quick-  $80-$7F, ext to 32 bits
-8-bit-displacement- disp fom PC, if 0, disp in extension word
-exg-mode-  01000=D<>D 01001=A<>A  10001=A<>D
condition
  0000 T     0100 CC HS    1000 VC     1100 GE
  0001 F     0101 CS LO    1001 VS     1101 LT
  0010 HI    0110 NE       1010 PL     1110 GT
  0011 LS    0111 EQ       1011 MI     1111 LE
EA-mode + EA-reg
   000 nnn Dn      100 nnn -(An)      111 001 xxx.L
   001 nnn An      101 nnn d(An)      111 010 d(PC)
   010 nnn (An)    110 nnn d(An,Xm.y) 111 011 d(PC,An)
   011 nnn (An)+   111 000 xxx.W      111 100 #xxx SR CCR

   ext for d(An,Xm.y)  bit 15     0 X=D  1 X=A
           d(PC,Xm.y)      14-12  mmm
                           11     0 .W   1 .L
                           10-8   000
                           7-0    displacement $80-$7F

mc word order:
   words,longwords in ext have msb/msw first
   opcode word comes first
     then, 1st EA - 1 or 2 words
     then, 2nd EA - 1 or 2 words
   byte exts are word length, with msbyte zero filled
```

1.130 primer

A Beginner's Guide to Assembly Language Programming with Tandem

Welcome to the beginner's guide! Your lessons await....

1.131 less1

Lesson 1 - Introduction

The manual should be used sequentially - the presentation is designed for "inductive" learning by absorption. The main section of this guide, i.e.

Tandem User Manual
is by contrast a complete exposition of
Tandem, which assumes some knowledge of assembly language.

This manual is written to give you sufficient intuitive grasp of assembly language to be able to go on to read the Amiga ROM Kernel manuals, and to be able to make a beginning at using Tandem to write and debug your own assembly language programs.

This manual should be used "hands on". You will need the following things:

- The Tandem program itself. Since you are looking at this, I presume you have got Tandem on disk and ready to run.

- An Amiga computer with 1 megabyte of memory, and workbench release 2.04 or higher. In practice, to do your own programming, you will really need a minimum setup of an Amiga 1200 with accelerator, a 68030, a 68882, a hard disk, a CD, and several megs of RAM. (and of course Amiga OS3.1).

- You must have following Amiga manuals:

  The Amiga Developer CD (published by Schatzruhe) - readily available

  Amiga ROM Kernal Reference Manual Libraries } Known as
  Amiga ROM Kernal Reference Manual Devices } RKM's
  Amiga ROM Kernal Reference Manual Includes and Autodocs }
  The Amiga DOS Manual

The RKM's are available in Australia from Amadeus Computers, (02) 9651 1711

There is no really quick road to learning - the idea is to read through this guide, and the above manuals, over and over as you absorb things.

I will assume you have some basic knowledge of using the CLI (Shell) and of BASIC programming. I will also assume you know in a general way about hexadecimal numbers, and ASCII. If you don't, then get from a library an introductory book about computers, which will explain about hexadecimal

numbers. You should also read about CHR$() and ASC() in the AmigaBASIC
manual, (or any other BASIC manual), for an explanation of ASCII.

1.132 less2

Lesson 2 - Your Working Environment

When Tandem first boots up, the window you see is Tandem's "Main" window.

I recommend that you let Tandem have its own private screen. To do so:

- click "prefs" on Tandem's main window
- click the "Screen Type" button to read "Non-lace"
- click "Save"

Then exit from Tandem and re-load it, to make the new "screen type" take
effect.

From then on, Tandem will open on its own screen, not on the workbench.
This will make the workbench less crowded.

However, the above is certainly not compulsory - yo can leave prefs as they
are, with Tandem opening on the workbench if you like, when just ignore the
rest of Lesson 2.

You can look at this guide by clicking "Guide" in the main window. However,
you will find it more convenient, when you are doing constant looking back
and forth, to use a setup like this:

1. press LeftAmiga/M to get back to the Amiga's Workbench Screen
2. click the Tandem.guide icon (in Tandem's drawer) to open Tandem.guide
   on the Workbench Screen
3. press LeftAmiga/M to switch back and forth between the Workbench screen
   and Tandem's scren. Click whichever to activate it if required.

Thus, you will only click Guide on Tandem's Main Window, for a quick glance.
But for sustained viewing, as when you use this Teaching Manual section,
the above setup with the guide on the workbench screen is the way to go.

(If you are a mouse lover, an alternative to 3. above is to drag the
workbench screen down as far as it will go, exposing Tandem's screen, and
drag it up again when you want to see the guide).

1.133 less3

Lesson 3 - A First program

On Tandem's "Main" window, click the "Edit sc" button. This will bring to
the front the "Edit" window. (You will see the dragbar of Main window poking
above the Edit window, so you can always bring it to the front by clicking
it, or pressing Esc, or selecting "Stop editing" on the menu).

On the Edit window you edit "sc" - that is, "source code". Source code, or sc as I will from now on abbreviate it, is the program you write. Tandem can at any time change your sc into "machine code", abbreviated to "mc", which is the finished product, a program that an Amiga can actually run.

Also on the edit window you will see a cursor, waiting anxiously for you to type something.

In assembly language, everything is numbered from 0 rather than 1. The lines are numbered in hexadecimal (hex) notation, so after line 0009 comes 000A, not 0010.

Basically, to create sc (=source code, remember), you simply type lines into the computer. You can use the backspace, delete and arrow keys in the normal way. There is a menu strip along the top, which shows the keyboard hotkeys for inserting and deleting lines, &c.

(e.g. to delete a line, the hotkey is Rt Amiga/D)

Begin by typing a line which reads:

```
       * A program to do nothing
```

The * in the first typeable column tells you that the line is a comment. That is, it is like a REM in Basic. An initial * or ; denotes a comment.

After typing the above, press <Return> to go to the 2nd line (line 0001), leave a space at the start of the line, and type the characters RTS, like this:

```
       * A program to do nothing
        RTS
```

n.b. The R of RTS in line 0001 is NOT below the * of line 0000!! It *MUST* have a space to its left - the R is typed in the 2nd typeable column, as above.

```
Wrong :(      * A program to do nothing
              RTS

Right :)      * A Program to do nothing
               RTS
```

The RTS is called an "opcode", and opcodes must NOT start in the 1st column. This rule will turn around and bite you until you get used to it.

1.134 less4

Lesson 4 - Assembling Your Program

Once your program reads as per the "Right" version, go back to the main window from the Edit window, by selecting "Stop editing" from its menu, or simply clicking its close gadget or presing Esc. On the main window, click "Assemble".

Tandem will change to a display reading "Assembling". Once "Pass 1" and "Pass 2" are finished, a message appears on the window, inviting you to click to acknowledge. If there are errors, for example if you misspelled RTS, then you must go back to the Edit display, where you can correct and try again. If you get a strange message about "no executable lines", you have left out the space before RTS.

If all goes well, there will be no errors, and after you click you will go to the main window, which will contain:

```
 0000             * A program to do nothing
 0000 4E75         RTS
 ENDI/END
```

This is called an "assembly list" for your program. The program's mc consists of the hex characters $4E75 - just 2 bytes, which tell the computer to "RTS".

Before you sneer at a program which does nothing, just reflect that many people have made a lot of money doing nothing. You're sitting on a goldmine.

Okay, you have created a program to do nothing. We'll test it in the next lesson.

1.135 less5

Lesson 5 - Test the Program Assembled in Lesson 4

The art of testing an mc program is called "debugging". The first thing you must always do, is to save the sc, before you begin to debug the mc.

1. Click "Edit" to go back to the Edit window. Thence, select "Save as" from the menu, and save the sc as:

   RAM:Nothing.asm (you need RAM: in the dir of the file requester)

   and your sc will be saved to RAM: disk. All sc files should always have .asm as a suffix. (obviously, "saving" to RAM: is not really saving at all; we're just doing it for illustrative purposes).

2. Now, press Esc to go back to the Main window, or click the Main window's drag bar. Thence click "Save mc" and save the mc as:

   RAM:Nothing

Now make the Amiga.guide window smaller, so you can see the Shell icon. Click the Shell icon top make a new Shell.

From that shell, enter the command:

   RAM:Nothing

and watch your program successfully do nothing! The program as you wrote it is not suitable to run as a workbench program started by clicking an icon,

but it runs ok from the CLI (shell).

Finally, zoom the Amiga.guide back to full size again.

Deleting and Inserting Lines

Now, click the "Edit sc" button on the Main window to get back to the Edit window. We will now create another program, which also does nothing. In fact, for quite a while our programs will do nothing. But they will do nothing in instructive and progressively more elaborate ways.

First, you can remove the old program by pressing several times:

  <Right Amiga> with D ("d" stands for "delete")

There is also a Menu item which you can select to delete a line.

To insert lines, you can press:

  <Right Amiga> with I ("i" stands for "insert")

There is also a Menu item which you can select to insert a line. Tandem's text editor works just like any other text editor.

With Tandem you can press the Help key at any time, to see "context sensitive online help".

1.136 less6

Lesson 6 - Teaching Programs

From now on, I will request you to load programs, rather than type them in. I have put them in a sub-dir within the Tandem dir, titled "Teaching".

For example, the program we already did is Teaching/1.asm which you could have loaded as follows:

1. Select "Load" from the menu of the Edit window
2. click the "Parent" button to get back the Tandem's drawer.
3. Click the "Teaching" drawer, then "1.asm" then the "Load" button.
4. Choose "Overlay previous memory contents"

And Teaching/1.asm would appear in the sc, replacing what was there before.

The Amiga's central heart contains:

- a "central processing unit" (CPU) called a Motorola 68000 (or, 68020, 68030 &c).
- a built-in clock that coordinates everything.
- a "bus" that the memory, disk, and other devices are attached to.
- 2 other CPU's called the "copper" and "blitter" which you don't program directly. These help run the picture and sound.

Subroutines

```
Now, load the sc for the program     Teaching/2.asm

It should be clear that you do that as follows:

   1. Select "Load" from the menu of the Edit window
   2. In the Teaching drawer, click   2.asm   then   Load
   3. Choose "Overlay previous memory contents"

n.b.: Fred and Jim start in the first column, like *, because they are not
opcodes but labels. Labels, such as Fred and Jim, must appear exactly the
same each time. Note carefully that these are 4 different labels:

    Fred  fred  fRed  FRED

The : which optionally follows a label is not part of the label.

Upper and lower cases don't matter for opcodes. You might have bsr or BSR,
it doesn't matter. But cases are significant for labels.

Now, assemble your program. If all goes well, you will see:

    0000 61000008      BSR Fred
    0004 61000006      BSR Jim
    0008 4E75          RTS
    000A              Fred:
    000A 4E75          RTS
    000C              Jim:
    000C 4E75          RTS

Instead of saving your sc and mc, we will "debug" it, using Tandem's
debugging facilities, in the next lesson.
```

1.137 less7

```
Lesson 7 - Single Stepping

The first thing to do when debugging a program, is to "single step" through
it. Above the bottom right of the screen, you will see something a bit like:

                    PC 00323490=line 000000      }The numbers 00323490 and
                    A7 level = 000001            }00323480 will vary from
                                                 }computer to computer.
      A7 00323480                                }

The things labelled D0 to D7, and A0 to A7, are called "registers". In this
particular program, we are especially interested in the register A7, which
is called the "stack pointer". The 00323480 in the above example tells me
that A7 has the contents 00323480, a hex number. The item "PC" is also a
register, called the "Program Counter". The program counter points to the
next item in your mc to be executed. We will now step our way through the
program:

To single step, press the <space> bar once. This will do the following:

    1. The PC will advance by 10  (in the above case, to 0032349A), and
```

point to line 000A. (labelled Fred also at 000A).

2. The SP (register A7) will reduce by 4 (in the above case, to 0032347C).

3. The A7 level will be 0002.

4. Line 000A will be complemented, to show that the PC points to it.

Put on your thinking cap, while I explain what happened inside the 680x0 chip which actually runs your Amiga. The registers are built-in to the 680x0, and are not the same as memory. Any opcode which must use memory takes longer than one which uses registers.

1. The PC was at the first opcode-containing line of your program, which happened in my case to be at the memory address 00323490. When your program begins, the Amiga's CLI puts the PC at its first line, and creates an area of memory for you called the "stack", and points the SP (i.e., A7) above the top of your stack.

2. The 680x0 waits for a pulse called a "T state" (=timer state) from the computer's inbuilt clock. When it gets a pulse, it grabs the next opcode the PC points to, which is 6100. The particular value 6100 is a BSR (i.e. "branch to subroutine"). Since 6100 is 2 bytes long, the PC now has the value 00323492. The 680x0's internal logic tells it that a 6100 opcode is followed by 2 more bytes, being the address in memory to jump to.

3. Next, the 680x0 grabbed the 2 bytes at 00323492, which were 0008. Instead of simply advancing to 00323494, the 68000 responds to the opcode by:

   i. adding 0008 to the PC value, making it 0032349A
   ii. pushing the value it would otherwise have had, i.e. 00323494, onto the stack.

4. Now, strangely enough, the stack builds downwards, so the SP, i.e. the register A7, gets the value 00323494 pushed down below its value. A7 first decreases by 4 (since 00323494 is 4 bytes long), and at its new address, i.e. 0032347C, it will contain the value 00323494.

You will notice that your labels are ignored by the 680x0. The only thing the 680x0 understands is opcodes, and the things like the above 0008 after them which are called "extensions". All opcodes are 2 bytes long: 2 bytes are called "words". The PC and other registers are 4 bytes long: 4 bytes are called "longwords". I could state what happened as follows, bearing in mind that it is correct usage to put a $ before hex addresses to distinguish hex numbers from decimal numbers. I will drop the leading 00's from the addresses:

1. The 680x0 took the 1-word opcode at address $323490. Because it had the value $6100, the 680x0 expected a further word at $323492.

2. after taking the word $0008 at adress $323492, the 680x0:
   i. Added the $0008 to its PC $323492, making $32349A.
   ii. Pushed the longword value its PC would otherwise have had, i.e. $323494, onto the stack, making the SP now $32347C.

```
       Each stack level is 1 longword, so the stack depth changes from 1
       to 2.
```

Now, in order to check what went onto the stack, I can Click "Memory". Tandem then prompts me for a value to inspect, so I enter $32347C, i.e. the address of the SP at A7. Sure enough, at the top of the memory display at address $32347C I find: $323484 which is the return address as above. (you can also enter !A7 instead of $32347C, since the "Memory" requester allows "!A7" as input to mean "the address pointed to by A7"). While the memory is showing, you can click up and down arrows to move up and down through the memory; Tandem shows memory both as hex, and over at the right as ASCII, if it is printable. Careful with this: some Amigas crash if you look at address $DFF000, or certain other addresses.

When you step this program, your values in the PC and SP will be different, but they will change as above. Make sure you read the above carefully, and try to understand what is happening. The above is seminal knowledge!!

Click the long "OK" button to return to the Main window.

Now, we are ready to execute line 000A. So, we press the <space> bar again in order to step the PC again. Here is what happens:

1. The PC jumps back to $323494
2. The SP increases to $323480
3. The A7 level is back to 1
4. Tandem complements line 0004, to show it is next to be executed.

Here is what has happened:

1. The PC grabbed the 1-word opcode at $32349A, which was $4E75. This is an RTS (return from subroutine).

2. The PC discarded its new value, i.e. $32349C, and instead popped a return address off the stack. On top of the stack was the longword $323494, so that is where the PC now points.

3. Tandem has highlighted line 0004, because it is at address $323494.

Well, that seems rather complex. Here is an AmigaBASIC program equivalent to your program, so you can follow its logic:

```
  10 REM A (bugged) program to demonstrate subroutines
  20 GOSUB 50
  30 GOSUB 60
  40 RETURN
  50 RETURN
  60 RETURN
```

If you examine the above carefully, you will see that it has a bug. The line 40 will return an error, since it is a RETURN without a GOSUB. That is because BASIC programs are self-contained, but the Amiga system runs your programs as subroutines of the CLI (or a workbench process). So, the last thing to be executed in your mc programs is always an RTS back to wherever your program started up from.

If you step through all of your program, it will step as follows:

```
   0000  BSR Fred
   000A  RTS
   0004  BSR Jim
   000C  RTS
   0008  RTS
```

And finally:

```
   0000  BSR Fred
```

The last step is artificial. What actually happened, is that your program exited back to Tandem. The very top item in your stack (called the "root" of your stack) at level 1 is a return address to Tandem. So the RTS in line 0008 actually RTS'ed out of your program back to Tandem. Tandem then re-built your stack, and put the PC back at the start, in case you want to run it again.

You will see an item "Run mc" at the bottom of your screen. If you click that, nothing will seem to happen, since Tandem will run your program, until it returns, and when it does, Tandem will re-position your PC at the start and re-build your stack. But if your program ran from the CLI, then it would run once, and the CLI would discard it after it is run, and go on with something else.

Caution: single stepping and especially running can crash the computer. The computer can easily hang up or go bananas if you single step or run your program. You will crash the system many, many times, and have to reboot, as you learn assembly programming. Even experienced programmers frequently crash the system, or lose control of it, in the course of debugging their programs. For this reason: ALWAYS SAVE YOUR SC BEFORE YOU DO ANY DEBUGGING!! You will probably have to learn this the hard way. You've been warned!

I have now introduced most of the basic concepts of assembly language, viz:

- hexadecimal (hex) notation
- source code (sc) and machine code (mc)
- comments, opcodes, labels and addresses
- the memory
- the central processing unit (CPU), i.e. the 680x0 chip
- the stack
- the registers, expecially:
    - the stack pointer (SP)
    - the program counter (PC)
- the Amiga's operating systems (i.e. the CLI and workbench)

Of course, my treatment of these so far is superficial, to say the least. I will now consider the most common opcodes, and then discuss the various addressing modes. After that, we will be able to consider programs that actually do something.

1.138 less8

Lesson 8 - The MOVE Opcode

The commonest opcode is the "MOVE" opcode. Load and assemble this program:

    Teaching/3.asm

Now, you should be in the Main window, ready to step it through. Initially the D0 register will probably contain:

    D0 00000001

The MOVE in line 0000 has a suffix, called an "extension", of .L The .L means to move a longword. In line 0006 .W means to move a word, and in line 000A .B means to move a byte. The extension is optional, and if you omit the extension, then .W is understood. I recommend that you never omit the extension. Putting the wrong extension is a common source of bugs.

Step through line 0000. You will see:

    D0 22222222

So, the longword entirely replaces what was there before. Then, step line 0006. You will see that D0 now contains:

    D0 22223333

That is, the "least significant" word of D0 is overlaid by 3333, but because the MOVE was only of a word, the "most significant" word remains. Then, step line 000A. You will see:

    D0 222233AA

So with the .B extension, only the least significant byte is overlaid. Each byte is made up of 8 bits. Since a byte is 2 characters of hex, each character of hex is 4 bits; a group of 4 bits is sometimes called a "nybble". You should be able to work out that a word is 16 bits, and a longword is 32 bits. 680x0 Assembly language can work on longwords, words, bytes or individual bits. (rarely nybbles).

Now, step line 000E. You will see that the contents of register D0 are copied to D1. Get into the habit of reading the MOVE instruction like this, in your mind:

    MOVE.W D0,D1   = "word move to D1, D0"

That is, if there are 2 addresses, read the second one first. This will help you to understand 680x0 arithmetic when we come to it.

Addresses and Values

Now consider the next two instructions:

```
0010 MOVE.L #4,A0
0016 MOVE.L 4,A0
```

The # means "the value". So, if you step line 0010, you will see:

```
     A0 00000004
```

That is, A0 gets the value 4 as you would expect. But if you step through line 0016 you will see something like:

```
     A0 00200810
```

If you omit the #, then Tandem will load the longword stored at address 4, whatever it might be. Now, click "Memory", and enter the address 4. You will see that the longword at address 4 is the same as what line 0016 put in A0. You must carefully understand the difference between:

```
     MOVE.L #4,A0 ; = "longword move to A0 the value 4"
     MOVE.L 4,A0  ; = "longword move to A0 the contents of address 4"
```

You will find that many bugs in your programs are caused by omission of the #, until you get some experience.

1.139 less9

Lesson 9 - Overview of the Registers

The 680x0 has 18 registers available to "applications" programs:

```
   the "data" registers:     D0 to D7
   the "address" registers:  A0 to A7 (A7 is also SP, the stack pointer)
   the program counter:      PC
   the flag register:        CCR  (CCR stands for "condition code register")
```

I will cover the flag register under the chapter on Arithmetic. The flag register also has a section called the SR, which is used by the Amiga's operating system but not by the applications programs, and other hidden registers depending on the member of the 680x0 family used.

The data registers are meant to hold values, and the address registers are meant to hold addresses. In fact, there is nothing to force you to follow this rule, and indeed the Amiga's system does not follow it. However, note the following:

```
   1. Many more types of arithmetic can be done on data registers.
   2. Many more forms of addressing can be done on address registers.
   3. Data registers can have extension  .B .W or .L
   4. Address registers only have extension .W or .L
      In practice, only .L should be used for address registers.
```

Thus, the internal features of the 680x0 encourage you to use the D0-D7 registers as data, and the A0-A6 registers as addresses. The A7 (SP) register must in practice be used as an address, and for one purpose only, i.e. the stack. In the Amiga, we will later see that the A6 register also has a special use most of the time.

In the subroutines that you write, it is good practice to use the registers fairly consistently, for ease of understanding. For example, you will see a special use of A4 in later Teaching examples.

```
Word Alignment

The 68000 has one big disadvantage, and it is this: it cannot do
instructions on odd addresses, if they have .W or .L extensions. Later 680x0
can do so, but do so slowly, and you should in practice never use odd
addresses for .W or .L data, so that your programs can run on an Amiga 500.
This will be the commonest cause of bugs in your programs, until you get
used to it, as they will crash if you run them on an Amiga 500.

Consider these lines:

0000  move.w #3,d0
0002  move.w 3,d0

line 0000 is ok - it moves an odd value to d0. But if you attempt to execute
line 0002, you'll crash the Amiga 500's system, since it tries to do a .W
instruction on an odd address.

Here is another example:

0000  move.b 5,d0                [Note: if your Amiga has a 68020 or
0004  move.w 6,d0                 higher instead of a 68000, then .W or
0008  move.w 5,d0                 .L to an odd address will not crash. But
000C  move.l 5,d0                 it is slow, and should be avoided].

Line 0000 is ok, since it is a .B
Line 0004 is ok, since it is an even address
Line 0008 or 000C will crash 68000, since they do .W or .L on odd addresses.

The PC must always point to an even address, since it fetches a word at a
time. If you write your sc in such a way that it would assemble opcodes to
odd addresses, Tandem will issue the error:

   E 49 (fatal) mc has become unaligned

and the assembly will abort.
```

1.140 less10

```
Lesson 10 - Errors

Now load, and try to assemble   Teaching/4.asm   which deliberately contains
bugs.

Can you see the errors? Tandem will report 3 errors, and return you to the
Main window. There you will see the errors listed.

If you edit any erroneous line to correct it, Tandem will still show it in
the list as an "error", since it doesn't know if you have corrected sc,
until such times as you conduct another assembly. You can correct errors by
clicking their left-most 4 characters, which puts you into the edit window
with the cursor on the erroneous line, where you can correct it (Each time
you change the sc, it gets harder for Tandem to find erroneous lines in the
list, so if you make many changes, Tandem might lose the ability to find
your errors, when you will need to re-assemble).
```

n.b. if you correct all 3 errors, Tandem will then still create another error, so there are really 4 errors. What happens is this:

1. Tandem passes through your sc twice. The first time through, it picks up nearly all errors.
2. The second pass, it assembles lines with "forward references". e.g. consider the line:

   move.l fred,d0

   Tandem does not know where "fred" is in pass 1, so it saves this line for pass 2. Since fred doesn't occur anywhere, it will cause an error in pass 2. But, if there are any errors in pass 1, Tandem does not do pass 2. So, until pass 1 is error-free, pass 2 errors will not appear in the assembly list.

Can you suggest corrections?

Line 0001 missed its leading *
Line 0002 has an "undefined" label. To "correct", just delete it.
Line 0002 begins in the wrong column
Line 0003 is misspelled

Note that Tandem cannot pick up any bugs, other than assembly language syntax errors, no matter how obvious. For example, if you enter

```
 move.l 3,d0
```

Tandem will happily assemble it, and will crash if you try to single step it on a 68000. If you omit the RTS, then your program will never return if you run it, but will go off into cloud cuckoo land. It is your responsibility to avoid such errors. You'll make them, and gradually learn through much pain and anguish to avoid them. And you will also learn through aversion therapy to save your sc before you start debugging your mc.

You will gradually build up a collection of subroutines that work, to incorporate into new programs. New projects usually begin with your writing a central core, debugging it, and then adding features one by one. It is usually a mistake to write a big program all at once. I always write a detailed user manual before I begin writing a program, and that way I know what I want it to do. So, for example, I wrote a draft of this Tandem.guide before I wrote Tandem itself.

1.141 less11

Lesson 11 - Pseudo Ops

Not all lines are opcodes or comments. There can also be lines called "pseudo ops" (so called, because they look like opcodes). The first line of your program that actually assembles something must be an "actual op", i.e. an opcode. There are two main pseudo ops:

```
 DS   equivalent to variables in Basic
 DC   equivalent to data in Basic
```

It is usual but not compulsory to label DC and DS statements. Load and assemble Teaching/5.asm

Notice that line 0006 contains a comment. You can put a comment after any line, with a leading semicolon. A whole line can be a comment with a semicolon instead of an asterisk, too. But an asterisk only starts a comment if it is the leftmost character.

The DC statement defines a constant. Line 0014 sets aside a longword in memory, which contains the value 4. "DC" stands for "define constant". But line 0018 simply skips a longword, leaving it with random initial contents. So, the first thing you must do with a DS is to move something into it, before you move anything out of it. But you can move things out of DC right away. I personally never change the contents of DC memory; but there is no rule against doing so. "DS" stands for "define space".

Actually, most assemblers including Tandem fill DS's with zeroes; but you should never assume initial values for DS's.

In the opinion of all good programmers everywhere, the only addresses your program should ever change are the contents of DS and possibly DC statements. Any other changes should only be done through the Amiga operating system in Amiga-sanctioned ways. I wouldn't even change DC statements, myself. Above all, you should never write programs that poke values into the addresses of opcodes. e.g.:

```
fred: move.l d0,fred+2 ;never do this! self modifying code is an abomination
```

But you can of course modify DS's, that's what they're for. Before you step the above program, click "View mem" to look at fred and bill. You will find that fred contains $00000004, and bill is regarded as random.

Line 0000 MOVEs to D0 the contents of fred, i.e. D0 00000004
Line 0006 MOVEs to bill from D0, so View mem will show bill now holds 4
Line 000C MOVEs to D1 from bill. Since bill is 4, D1 will be 4

Notice it would be a blunder to move from bill before you move to it. Study:

```
0000 Adc: DC.W 5
0001 Ads: DS.W 5
```

the first is 1 word long, with the value 5. The second is 5 words long, with initial random values. Seminal knowledge! You will get baffling bugs if you use DS where DC should be, or vice versa.

These are the commonest bugs for beginners:

- :(mixing up DC and DS
- :(omitting # before values
- :(unbalanced pushing & popping to stack (not covered yet)
- :(omitting extensions
- :(reversing the order of ADD,SUB & CMP addresses (not covered yet)
- :(leaving the (A6) of JSR _LVO's (not covered yet)
- :(changing DBRA counters (not covered yet)
- :(getting mixed up which registers contain what

1.142 less12

```
Lesson 12 - More on the stack

Often, you will want to temporarily store something, but not go to the
trouble of making another DS to store it. You can use the stack to store
register contents temporarily. Only use .L extensions when you do so!

Load and assemble Teaching/6.asm, and step it through. After line 0000, you
will see:

 D0 12345678

After line 0006, A7 will reduce by 4, and A7 level will be 2. That is:

 -(a7) means to the 680x0: reduce A7 by 4, then move the value there.
 (a7)+ means to the 680x0: move the value to A7, then add 4 to A7.

Note how -(a7) reduces the value first, then pushes. But (a7)+ pops first,
then adds to the value. If you think very carefully about this, you will see
that this makes things convenient (but confusing at first) to programmers.

Now consider this:

0000 * don't run this program!
0001
0002  move.l #$12345678,d0
0003  move.l d0,-(a7)
0004  rts

The above  program contains a bug: it pushes a value onto the stack, but
then RTS's, which pops the value off the stack and tries to jump to it. This
is absolutely catastrophic. One of the most important aspects of programming
is to ensure that within a subroutine, your pushes match your pops. For this
purpose, you should keep the flow of your subroutines clear and simple. A
common mistake of style for beginners, is to write subroutines with complex
jumping about like a bowl of spaghetti.

It is a good idea to follow these rules (some of which you won't understand
on your first run through this manual):

1. Use registers for consistent purposes within a subroutine, if possible.
2. Use comments, particularly to explain the use of registers and CCR flags.
3. Try to put addresses in A regs and values in D regs.
4. Keep the stack longword aligned (treat this as compulsory!).
5. Never have multiple entry points to a subroutine.
6. Try to avoid having multiple exit points from a subroutine, except that
   it is ok to have a "good" RTS and a "bad" RTS. Good RTS's can return
   NE, and bad RTS's with EQ.
7. Try to follow a consistent rule: that subroutines leave all regs
   unchanged, except those that return a result to the caller.
8. Use JSR only for library calls
9. Never use JMP
10. Never BRA to a subroutine instead of BSR followed by RTS.
11. Don't BRA to the middle of a DBRA loop, or change its control variable.
12. Don't pop return addresses off the stack or calculate addresses to jump
```

```
   to or other such muddly things.
13. In general, keep things simple, well documented, and free of kludges.
```

1.143 less13

```
Lesson 13 - Arithmetic

Before going on, please change one of the internal parameters of Tandem, as
follows:

  - on the Main window, select "Sundry"
  - click "Maximum mc bytes/line on assembly list"
  - enter 4 for "max bytes shown"
  - click "Use"

The above lets you see more sc (but less mc) on the assembly list, so you
can see sc lines without the comments being chopped off.

The 680x0 has many forms of arithmetic it can do. Assemble Teaching/7.asm
and on the debug display you will see something like:

    NX CC EQ MI VC

These 5 values are called "flags", and together they make up the useful bits
to you of a special register called the CCR (condition code register).

These are the possible values of the 5 CCR flag bits:

    NX or X   ("Extend" or "Not extend")
    CC or CS  ("Carry clear" or "Carry set")
    EQ or NE  ("Equal" or "Not equal")
    VC or VS  ("Overflow clear" or "Overflow set")
    PL or MI  ("Plus" or "Minus")

I will explain X and NX later - they are rather odd. But the others are
traditional to all CPUs. Here is how they work:

1. MOVE clears V and C to VC and CC, and sets EQ/NE PL/MI as per arithmetic.
2. Arithetic sets:

   EQ if a result is zero, else NE                          the zero flag
   PL if a result is +ve, else MI                           the sign flag
   CC if a result is ok for unsigned arithmetic, else CS    the carry flag
   VC if a result is ok for signed arithemtic, else VS      the overflow flag

Now, you will need to understand what values a byte can contain:

1. A byte can be from 0 ($00) to 255 ($FF).
2. So, the unsigned value of a byte is from 0 to 255.
3. But a byte can also be interpreted to have a sign. It can hold a value
   from -128 to +127. If its most significant bit is 0, the byte is
   positive. If 1, the byte is negative.
4. So, in a signed byte:
    Values 0 to 127 ($00 to $7F) are the same as unsigned.
    Values 128 to 255 ($80 to $FF) are negative. $80 is interpreted as -128.
```

```
      Then, $81 as -127, $82 as -126, and so on until $FF as -1.

5. The 680x0 does not "know" whether you are doing signed or unsigned
   arithmetic. It always sets all the flags, and you use the ones relevant.

Now, step through the program.

0000  move.b #$7F,d0    sets PL (since $7F is +ve in signed arithmetic)
                             NE (since $7F is non-zero)
                             CC and VC  (MOVE always clears V and C)

0004  add.b #2,d0

      this leaves   D0 ......81

      In unsigned arithmetic, this is:  127+2=129  which is ok.
      In signed arithmetic, this is:    127+2=-127 which is bad

      So, the flags are:     MI  (since $81 is -ve in signed arithmetic)
                             NE  (since $81 is non-zero)
                             CC  (since the result is ok in unsigned arith)
                             VS  (since the result is bad in signed arith)

Now, step lines 0008 and 000C. They are equivalent to:

      Unsigned:     129-2=127 which is ok    hence CC
      Signed:      -127-2=127 which is bad   hence VS

Now, do lines 0010 and 0014. Equivalent to:

      Unsigned:       1-2=255 which is bad   hence CS
      Signed:         1-2=-1  which is ok    hence VC

Finally do lines 0018 and 001C. Equivalent to:

      Unsigned:     255+2=1   which is bad   hence CS
                     -1+2=1   which is ok    hence VC

So, to sum up: if arithmetic crosses the $80 value, it is bad for signed
arithmetic (since +127 jumps to -128), so you get VS.

And if arithmetic crosses the $00 value, it is bad for unsigned arithmetic,
(since 255 jumps to 0), so you get CS.

The result CS reminds you that a math result is bad for unsigned arithmetic,
while VS reminds you the result is bad for signed. If you are working on
.W (word) extensions, the flags are set if you cross $8000 and $0000. And
for .L, $80000000 and $00000000. You can also multiply and divide, but in
assembly language, ADD and SUB are much more common.
```

1.144 less14

```
Lesson 14 - Program Branching

The results of arithmetic are often used to control the flow of programs.
```

```
Load and assemble  Teaching/8.asm

The instruction "BRA" means "branch always". This program jumps about like
this:

   0000 0006 000C 000A return

Consider this AMIGABasic program:

 0 REM Jump about
10 LET A=20
20 IF A=20 THEN 40 ELSE 50
30 REM: can't get here!
40 STOP "Equal"
50 STOP "Not equal"
60 END

This program will stop at line 40. But if line 10 read,  say, A=30, then it
would stop at line 50. You can see how line 20 uses arithmetic to control
program flow. Now here is the assembly language equivalent:

Load, assemble and step through  Teaching/9.asm

The CMP stands for "Compare". It sets the flags as if it had subtracted the
second address from the first. Yes! that way around - unfortunately - that
is why it is read:  "compare with d0, the value 20". For example, if d0 were
30, it would be PL (since 30-20>0) but if d0 were 10, it would be MI (since
10-20<0).

I should stress that the SUB does not actually take place in a CMP; it is
just that the flags are set {i}as if a SUB had taken place, but in fact
the 2nd address (D0 or whatever) remains unchanged.

The BEQ stands for "branch if equal". Depending on what is put in D0 in
line 0001, the program will return at line 0005 or 0007. A list of branches:

               BRA  branch always   (also written BT for Branch True)
               BEQ  branch if equal
               BNE  branch if not equal

signed:                                unsigned:
   BVC  branch if overflow clear          BCC  branch if carry clear
   BVS  branch if overflow set            BCS  branch if carry set
   BLE  branch if less than or equal      BLS  branch if lower or same
   BGE  branch if greater than or equal   BHS  branch if higher or same
   BLT  branch if less than               BLO  branch if lower
   BGT  branch if greater than            BHI  branch if higher
   BPL  branch if plus
   BMI  branch if minus

In all cases except BRA, you do arithemtic (e.g. ADD,SUB or CMP) and then
the flags will determine how the Bcc such as BEQ jump.
```

1.145 less15

```
Lesson 15 - An Example of Conditional Branching

Try   Teaching/10.asm

The item 'a' is equivalent to ASC("a") in Basic, i.e. it is 97 (=$61). Step
the program through - watch the CC/CS flag - and re-assemble with various
things in line 0002. It is difficult to follow exactly how this works, but
puzzle it through, and you'll learn a good deal. Writing assembly language
is admittedly rather difficult.

Here is an equivalent Basic program, but it's also hard to follow:

 10 REM Subroutine to convert a-z to A-Z  (without using UCASE$)
 20 LET D0=ASC("c")   :REM Try various characters in place of "c"
 30 GOSUB 60
 40 PRINT "The upper case is ";CHR$(D0)
 50 END
 60   REM If D0=a-z, convert to A-Z
 70   IF (D0>=ASC("a")) AND (D0<(ASC("z")+1)) THEN
 80     LET D0=D0+(ASC("A")-ASC("a"))
 90     ENDIF
100   RETURN
```

1.146 less16

```
Lesson 16 - Decrement Bra Conditional

As well as BRA for branches, you can have DBRA, which is like FOR..NEXT in
Basic. Consider this AMIGABasic program:

10 REM demonstrate a FOR...NEXT loop
20 LET D0=0
30 FOR D1=9 to 0 step -1
40 LET D0=D0+10
50 NEXT D1
60 STOP D0
70 END

It multiplies 10 by 10, and prints 100. Now load and assemble the assembly
language equivalent in  Teaching/11.asm

Step it through. That will take a while, since it loops around from 0010 to
000A 10 times. What happens at line 0010 is this:

 1. first, D1 is decremented by 1
 2. if D1 <> -1, a BRA takes place (in this case, to Loop, i.e. line 000A).
 3. but if D1 = -1, no BRA takes place, so the PC falls through to line 0014

You can use any register from D0 to D7 to control the DBRA loop. It is not
good programming practice to change D1 within the loop. Also, it is bad
practice to jump into the middle of a DBRA loop; it is quite ok to BRA out
of the loop though. After the loop falls through, the control register (D1
in the above case) will have the value $FFFF. To do a thing X times, you
start with X-1 in the control register. Note that a loop always executes at
```

least once. The 680x0 has no way of knowing that you are in the midst of a DBRA loop; it will only know once it comes to the DBRA. Remember, best practice is to enter a DBRA loop only by falling through the label at the start. (n.b. DBRA does not change any CCR flags).

There are two other types of branches:

JSR Jump to Subroutine: this is used to call subroutines outside the scope of your program: for example, the subroutines in the Amiga ROM Kernal. But for your own programs, use BSR to call internal subroutines.

JMP Jump: this works like BRA, but has more options. In my opinion, a well written program never uses this instruction. The only good use for JMP is in system "vectors", which are beyond the scope of this manual (applications programs should not use system vectors).

1.147 less17

Lesson 17 - Logical Operations

Load and assemble Teaching/12.asm. The MOVEQ opcode is fast & compact alternative to MOVE.L #xxx,Dn. e.g.:

```
 MOVEQ #1,D0     is equivalent to  MOVE.L #1,D0
```

but operates much quicker. So why not use MOVEQ all the time? The catch is, that the value moved into the D register must be from -128 to +127. In this case, we are moving +1, so we use MOVEQ, as it is in the range.

The AND operation pairs up the bits in two registers, and does a "logical and" on them. Consider this example, of an AND.B

```
    sending register:                %01001100
    receiving register:              %01000101

    new value of receiving register: %01000100
```

You will see that, as I have written out the value in binary, I prefix a %. Hex numbers get a leading $, and binary a leading %. In the above example, only in those bits where there is 1 in both the sender and receiver does the receiver get 1. Here is an OR.B

```
         sender       %01001100
         receiver     %01001010

         result       %01001110
```

As you can see, the receiver gets a 1 wherever either the sender of receiver (or both) have a 1.

Here is an EOR.B (EOR stands for "exclusive or")

```
         sender       %01001100
         receiver     %00101010
```

```
         result       %01100110  (note how receiving 1's flip sending bits)
```

This puts a 1 into those places that have a 1 in either the sender or receiver but not both. Try the program with various inputs, and AND, OR or EOR, until you get the idea.

1.148 less18

```
                Lesson 18 - Shifts and Rotations
```

The 680x0 can shift all the bits in a D register or address around in a cirle, or back and forth. Load and assemble Teaching/13.asm

line 0002 changes $44442222 to $22221111

Shifting binary numbers 1 space to the right divides them by 2, just as shifting decimal numbers 1 space right divides them by 10. LSR.L #1,d0 means "longword logical shift right D0 by 1 step". You can rotate or shift D registers up to 31 steps left or right. ROL.L #4,D0 means "longword rotate left D0, 4 steps". You will see that this rotates the digits around - they all rotate around like a barrel. But a shift brings 0's in one end, and drops out whatever is at the other end. The last 1 or 0 to fall out of whatever is shifted or rotated causes CS or CC respectively. A 68020+ does shifts much faster than a 68000.

You can also do ASR, arithmetic shift right, which instead of filling the left hand bit with zeroes, duplicates whatever was in the leftmost bit. This causes a signed division by 2. ASL works like LSL. There is also ROXL and ROXR, which include the X in the rotation. Any elementary computer book will tell you more about shifts.

```
                Assmbly Language
                 has
diagrams showing the various shifts.
```

1.149 less19

```
 Lesson 19 - Addressing Modes
```

I have covered most of the opcodes of 68000 assembly language in at least a cursory way. I will now mention the addressing modes. Load and assemble Teaching/14.asm and fasten your seatbelt for some heavy going.

Notice the DC.B fred. As well a list of 1 byte values, a DC.B can contain a 'string', i.e. text between apostrophes. Within a string, consecutive apostrophes stand for a single enclosed apostrophe. Compare:

```
assembler      dc.b 'Hello!',0   ;some assemblers also allow "Hello!"
BASIC          "Hello!"
```

Usually, DC.B strings have a 0 after them, to show that thay have finished. It is also usual to put a DS.W 0 after any DC.B lists which contain strings, since this causes the subsequent lines to be word aligned, just in case the

strings took up an odd number of bytes. DS.W and DS.L always skip to an even address if they're at an odd address. Tandem assembles fred as:

```
  48 65 6C 6C 6F 21 00   and the DS.W skips an extra byte to make it even.
```

The instruction:

```
  LEA fred,a0
```

points A0 to fred. The instruction LEA stands for "Load effective address". It works like MOVE.L #fred,A0 but is quicker.

Now, A0 is pointing to fred. Subsequent lines will look at parts of fred. If you click View mem and specify fred you will see that it is at the address A0 points to, and that it contains 48656C6C 6F2100. The address fred is called "Absolute Long Address" mode.

Now step line 0006 MOVE.L A0,D0 This merely moves A0 to D0. The address A0 is called "Register Direct address" mode.

Now step line 0008 MOVE.L (A0),D1 This moves the contents of memory address A0 to D1. The extension is .L so a longword was moved. This is an "indirect" address, since A0 points to the address. But A0 without parentheses is a "direct" address, since A0 itself was moved in line 0003 whereas in line 0004 the contents of the address pointed to by A0 was moved to D1. Now, A0 happened to point to fred, so we can see that the first longword of fred, i.e. 48656C6C now appears in D1.

Now step line 000A MOVE.B 1(A0),D2 The address 1(A0) is called "address register indirect with displacement". It works like this: First, the 680x0 takes A0, and then adds 1. It then retrieves whatever is at that address, and puts it in D2. A0 does not get changed. The displacement (also sometimes called an "index") can be from -32768 (i.e. $8000) to +32767 (i.e. $7FFF).

Now A0 was at address fred, so 1(A0) will get the 2nd byte of fred, i.e. 65. Notice that if A0 is even, MOVE.W 1(A0) would crash a 68000, since A0+1 is odd, and the .W extension cannot operate on odd addresses.

Now step line 000E MOVEQ #2,D3 The address #2 is called "immediate data" address mode, since the value moved is built-in to the mc. It puts the value $00000002 into D3.

The next one is very complex: 0010 MOVE.B 1(A0,D3.W),D4 The address 1(A0,D3.W) is called "address register indirect with index" addressing mode. It works like this:

- the 680x0 begins with A0
- it then adds D3.W
- finally, it adds the index 1

The 1st register is always an A register. A0 in this case. The 2nd can be a D register or an A register - in this cased, D3. Its value is can be .W or .L. So, D3.W can be $8000 to $7FFF, i.e. -32768 to 32767. But .L can add anything from $80000000 to $7FFFFFFF to A0. Finally, the index can be anything from $80 to $7F, i.e. -128 to +127. In the above case, D3 was 2, and the index was 1, and A0 was fred. So we looked at: fred+2+1, i.e. fred+3 which contained $6C. So, sure enough, we see 6C in D4. Whew! Incidentally,

you don't need to know the names of the addressing modes, just how they work. As a beginner programmer, you won't use address register indirect with index much, until you get some experience with simpler addressing modes.

the 68020+ contain even more complex addressing modes, but these are rarely used, even by experienced programmers. They just make assemblers hard to write!

The final address mode in the program is line 001C MOVE.L fred,D5 which is called "absolute long address". theoretically, you could have fred.W, if you knew that fred was in a memory address from $FFFF8000 to $00007FFF, but in the Amiga you never know this, except in 1 special case I mention later. So, if you leave off the extension from fred, Tandem assumes .L

Note that in the above, I am referring to the address itself (i.e. fred) as fred.W or fred.L, not what is moved in or out of it. What is moved in ot out of the address may be .B, .W or .L, with default .W

There are 2 other addressing modes: PC relative, and PC relative with index. These operate like d(An) and d(An,Xn.y), but with PC in the place of the A register. However these are advanced usage. I already mentioned previously the addressing modes -(An) and (An)+, which are used for pushing and popping values to and from the stack. Here is a summary of all the addressing modes:

```
Dn or An    register direct    e.g.  A4  D3
            the contents of the register

(An)        address register indirect  e.g.  (A5)
            the address pointed to by the register

-(An)       address register indirect with predecrement  e.g. -(A0)
            reduce the register first, then the address pointed to

(An)+       address register indirect with postincrement  e.g. (A5)+
            the address pointed to first, then increase the register

d(An)       address register indirect with displacement  e.g.  30(A5)
            the address is calculated by An+d. d can be $8000 to $7FFF

d(An,Xm.y)  address register indirect with index e.g. 5(A4,D3.W) -2(A2,A1.L)
            the address calulated by An+Xm+d. d can be $80 to $7F. X can be
            D or A. .y can be .W or .L

xxx.y       address absolute.   e.g.  4  fred+17
            The address pointed to by the value xxx   .y can be .W or .L,
            but .W is not used in the Amiga. Tandem defaults to xxx.L

d(PC)       program counter with displacement   e.g.  40(PC)
            The contents of the current address+d. d can be $8000 to $7FFF

d(Pc,Xm.y)  program counter with index   e.g. 5(PC,D3.W) -2(PC,A1.L)
            Like d(An,Xm.y) but PC replaces An.

#xxx        immediate data    e.g.  #4  #fred+16
            the value xxx  (not the contents of address xxx)
```

I have now covered all the most important addressing modes and opcodes. I

will now progress to programs that actually do something.

1.150 less20

Lesson 20 - Some Homework

At this stage, you have a general sort of sketchy overview of 680x0 assembly language, and of how to use Tandem. Before progressing to Part 2 of this manual, you will need to do the following:

1. Read, and re-read,
   Tandem Manual
   . You should hopefully
   understand most of it. (the material about MACRO's is hardest).

2. Buy or borrow from a library a book about 68000 assembly language. Most Amiga dealers have a book in stock about "Amiga Machine Code". Don't worry if it seems to be out of date.

Read the above several times. Don't necessarily try to understand everything in detail, just what you can.

For the rest of the lessons, you will need the Amiga ROM Kernal Manuals.

1.151 less21

Lesson 21 - Calling Amiga's ROM Libraries

Load and assemble Teaching/15.asm It is a program that actually does something, but not much admittedly.

This will take a long while to explain. Before you step it through, I will explain it as well as I can. This knowledge is very vital, so concentrate!

1. To do anything useful on the Amiga, you must make ROM library calls. The Amiga has zillions of subroutines built-in to permanent read-only memory, called ROM. the basic role of the programmer is to make ROM calls.

2. When your program is running, it is said to be a "task" - in fact, part of a special type of task, known as a "process". A process is a task that is allowed access to diskfiles. You should always assume that there are several tasks and processes running simultaneously in the Amiga with your process. You have to follow various rules, in order to stop these tasks and processes fouling each other up.

3. Almost any program (including Teaching/15.asm) spends most of its time waiting. The Amiga takes advantage of this, by doing "multitasking". It works like this:

   * Your program runs, until it has to wait for something.
   * When it does, the computer puts your task to sleep, until something happens. While it sleeps, another task runs.

* Then, when whatever it was happens, the other task goes to sleep, and your task wakes up.

Thus, there can be several tasks running, all apparently at full speed.

4. Another thing you need to know is that the 680x0 is not the only central processing unit (CPU) in the computer. There are 2 others, called the "copper" and the "blitter". You generally do not need to program these - the Amiga's system does it for you, based on your ROM Library calls. For example, the call _LVODisplayBeep in the 15.asm programs the blitter to beep the screen. While that happens, your process sleeps.

5. The first thing your program does, is to open a library, called the "intuition.library". Why does it do that? Because every ROM library call belongs to a library. In the RKM "Includes and Autodocs", you will see the main section is hundreds of ROM library calls, divided up into libraries. The master library is "exec.library" - it is always open, and it opens and closes other libraries for you, and does other fundamental things. The libraries each have special areas of operation. Intuition.library specialises in screens and windows. Under it, I found the call DisplayBeep, which is just what I needed. So, here is how to open intuition library:

```
move.l 4,a6
lea intname,a1
moveq #37,d0
jsr -72(a6)

intname: dc.b 'intuition.library',0
```

6. The Autodoc for OpenLibrary says you must:

- point A1 to a null terminated string with the library name
- make D0 the minimum acceptable version. Version 37 is release 2.04 Version 40 is release 3.1 (no programs are written any more for Release 1.3, which should be considered obsolete). So D0 is always 37 or 40.
- To call OpenLibrary, jump to an address 72 bytes behind the base of the exec.library
- All library calls must have the library base in A6. The base of the exec.library is stored in address 4, the only fixed memory address.

7. Now in practice you do not write the open library routine like under 5 above. It is bad practice to put "magic numbers" like 4 and -72 in your programs. The Amiga operating system has symbolic names for everything, and your programs should use them. The 4, which is the address where the address of the exec.library base is stored, is the only fixed address in the Amiga software. This address is called:

```
_AbsExecBase
```

instead of putting: `_AbsExecBase: EQU 4` you should put:

```
XREF _AbsExecBase
```

and the -72 means that the ROM call OpenLibrary is 72 bytes below _AbsExecBase. When you make library calls, the Amiga expects you to put

_LVO as a prefix before them. Instead of putting _LVOOpenLibrary EQU -72 you put:

```
       XREF _LVOOpenLibrary
```

and Tandem finds from a special file of constants where OpenLibrary is (i.e. at -72). All ROM library call addresses (called "offsets" because they are relative to the library bases) are resolved by Tandem at assembly time from XREF statements. Here then is the correct way to call a library of release 2.04 or later, which is called version 37:

```
       XREF _AbsExecBase       ;}these should appear
       XREF _LVOOpenLibrary    ;}at the start of the program

      intname: dc.b 'intuition.library',0 ;library name
       ds.w 0                  ;align sc
       .......

       move.l _AbsExecBase,a6  ;point A6 to base of exec.library
       lea intname,a1          ;point A1 to the name of intuition.library
       moveq #37,d0            ;at least version 37, i.e. release 2.04
       jsr _LVOOpenLibrary(a6) ;open intuition.library
```

8. The Autodoc for Open Library tells you that the result returns in D0. So, you should make a DS.L to put the library base you have opened. In this case, it is intuition.library, and a suitable name for its base is:

```
     intbase: ds.l 1
```

So, after you call OpenLibrary pointing to intname, you would get in D0 the address of IntuitionBase, which you place in intbase. If the library won't open (shouldn't happen if you have release 2.04 or higher) then your program must abort, because you cannot proceed if you can't open the libraries you need. If D0 returns null then OpenLibrary failed. Here is the relevent code:

```
     jsr _LVOOpenLibrary(a6)  ;open intuition.library
     move.l d0,intbase        ;save intbase for later use in A6
     beq Abort                ;abort if bad
     ......

    Abort:
     moveq #-1,d0
     rts
```

You will see that if D0 is null MOVE.L D0,intbase will set EQ. This makes your program fail, so I created a label Abort and put -1 into D0 and returned to the CLI with RTS. A return value of 0 in D0 tells the CLI your program executed ok. A value of -1 (=255) is a catastrophic failure (see the AmigaDOS manual under "FAILAT").

9. Now that you have a pointer to intbase, you can do your DisplayBeep. The Autodoc for DisplayBeep tells you to make A0 null to beep all screens. Here is the relevant code:

```
    XREF _LVODisplayBeep
    ......
```

```
        move.l intbase,A6       ;get IntuitionBase
        sub.l A0,A0             ;make A0 null to beep all screens
        jsr _LVODisplayBeep(A6) ;do the beeping
```

10. The Amiga will take care of forbidding all other tasks from using the blitter while it uses the blitter to do your DisplayBeep. Other tasks may be using the intuition.library, but the Amiga allocates everything correctly.

11. Finally, whatever your program opens or allocates, it *must* close or de-allocate, so that other tasks can use them. So, you must close intuition.library seeing that you opened it. Of course, if your program jumps to Abort it does not close intuition.library, since it was never successfully opened. CloseLibrary is in exec.library, so we need _AbsExecBase back in A6. The Autodoc for CloseLibrary tells us to put the library base we want to close in A1. This routine cannot fail, as long as it is called correctly. Here is the code:

```
        xref _LVOCloseLIbrary
        .....

        move.l _AbsExecBase,a6     ;point to exec.library
        move.l intbase,a1          ;base of library to be closed
        jsr _LVOCloseLibrary(a6)   ;close intuition.library
```

12. Finally, we go back to the CLI. We set D0 to null indicating successful completion. Here is the code:

```
        moveq #0,d0  ;indicate success
        rts          ;return to the CLI
```

Now, keep reviewing the above 12 points, and referring to the program, until you start to understand. Finally, assemble and step your program through, to see that it works. If it does, then from the Main window click "Save mc" and save your mc as "RAM:Beep". Then, go to the workbench, open another CLI, pull down the screen a bit so you can see both screens, and enter the command: RAM:Beep and the screens should both beep. Success!

1.152 less22

Lesson 22 - Introduction to MACRO's

First, load & assemble 16.asm. Now 16.asm does the same as 15.asm, but it uses MACRO's. First look at the Program subroutine.

In Program, you will see the following, which look like actual ops:

```
     intopen
     intclos
     beep
```

but in fact they are called "MACRO references". Here is how it works:

1. At the start of your program, are 3 "MACRO definitions". A MACRO

definition is a list of sc lines, between a MACRO pseudo op and an ENDM pseudo op. The assembler does not look at the lines between the MACRO and the ENDM. Instead, it skips all the lines, and remembers only the name of the MACRO, which is the label of the MACRO pseudo op.

2. Then, here and there in your sc, you can put MACRO references, which look like actual ops. When the assembler finds a MACRO referencem, it assembles all the lines between the MACRO and the ENDM as if they had between physically whre the MACRO reference was. Thus a MACRO can be assembled many times in a program. Note carefully how this happens, by pressing the "Halves" button, and putting the main & edit windows together, and scanning the sc and the mc. Do this until you understand what the assembler did. Seminal knowledge!

3. Note the difference between a MACRO and a subroutine. A MACRO is assembled each time it is referenced, at assembly time. But a Subroutine is assembled once, and is repeatedly called at run time. At first, you would think that subroutines were much more useful than MACRO's. And, of course they are. Subroutines make programs more compact. They are for the convenience of the computer. But MACRO's are for the convenience of the programmer. Look again the Program - note how its logic is more intuitive and clear than 15.asm.

4. Every programmer makes many "structures" in memory - and MACRO's make these structures easy to program and maintain.

1.153 less23

Lesson 23 - MACRO's with Parameters

Load and assemble 17.asm. The real power of MACRO's is found in its parameters. They work like this:

* When a MACRO definition is written, certain thinks are left "blank", with the details to be filled in as required. These are called "parameters". Parameters are numbered 1 to 9. The programmer the fills in the values of the parameters after the MACRO reference, like this:

  * the paramers are written \1,\2, &c in the MACRO definition.
  * then, in the MACRO reference, there is a list of strings (without apostrophes) in the address field, being the actual strings with which to replace the \1, \2 &c in the MACRO definition.

You can also read

MACRO
 & try to understand. Don't worry

at this stage about synthetic labels and NARG and \0 - you will pick these up later.

Like the last lesson, compare carefully the sc and the mc, and try to understand what the assembler has done. This is vital information.

1.154 less24

```
                Lesson 24 - Conditional Assembly

The idea of conditional assembly is that the lines between an IF and an
ENDC are assembled, only if the statement is true. The IF can take various
forms. e.g. consider this;

   fred: EQU 0             ;set 1 to assemble Fred

   IFNE fred               ;assemble this only if fred<>0
  Fred:
   moveq #0,d0
   rts
   ENDC

In the above, before you assemble, you can change the EQU at the start to
1, to force the assembly of the Fred subroutine. Thus, for example, you
could have a file with some subroutines, and set "switches" like fred above
to assemble those needed.

Read
                Conditional Assembly
                 to see the forms that IFcc can
take. Then load and assemble 18.asm, and see how it operates. Note the way
IFC does a character-by-character comparison between the parameter passed
and the second address. Keep comparing the sc and mc, seeing what gets
assembled and what doesn't, until you understand.
```

1.155 less25

```
Lesson 25 - NARG, Conditional Assembly II

Load & assemble 19.asm. The sum MACRO is rather complex - it assembles the
parameters - up to 5 - passed by the various MACRO references under "do some
sums". When a MACRO reference takes place, the assembler sets a label called
NARG equal to the number of parameters in the MACRO reference. So, in the
case of:

   sum d0,4,5

Tandem sets NARG=3, since therer are 3 parameters. Now in the MACRO, there
are 5 nested IFcc..ENDC's. The assembler will calculate the first 3 to be
true, since NARG-1/2/3 are GE, but NARG-4/5 are LT, so false. Thus, the
MACRO stops after adding the three terms set.

Like the last lesson, study carefully waht actually gets assembled by each
MACRO reference.
```

1.156 less26

Lesson 26 - Immediate Mode, Sundry opcodes

A very unusual aspect of Tandem is that it allows "Immediate mode" instructions, like in BASIC programming. You must first assemble something, so load and assemble 1.asm. Now, when mc exists, you can press the Immed button, and enter an immediate mode instruction. You can of course crash the system with immediate mode, and no doubt will do so. You get better at assembly language programming with practice, and immediate mode is a good way to get some practice. e.g. try these in immediate mode:

```
   moveq #2,d0
   moveq #3,d1
```

and watch D0 and D1 get the values you input. Don't put just anything in A7, or you'll most likely crash. Also, don't put values into memory, except for the address of any DS statements in your program. You can also make an area in stack to play with, like this:

```
   sub.l #64,a7        (make the number divisible by 4!)
```

And then you can poke things into values from (a7) to 63(a7) safely. Finally, enter:

```
   add.l #64,a7
```

to restore your stack. If your sc contains subrountines, you can bsr to them in immediate mode, and all sorts of other things. Go ahead and experiment - don't worry about crashing. When you crash the system, just reboot, and try and work out why you crashed it. You can try out other opcodes I haven't covered yet. From a library, get a book on 68000 assembly language and try out all the opcodes (not the priveleged ones, or TAS or LINK or UNLK).

e.g. try this:

```
  moveq #20,d0
  divu #4,d0
```

The divu divides the contents of d0 by 4, so the result will be $00000005. Now try this:

```
  moveq #21,d0
  divu #4,d0
```

and watch how the remainder gets put in the top half of d0. Then try:

```
  swap d0
```

and swap the halves, or

```
  exg d1,d2
```

to switch registers around &c. You can do a lot of self instruction this way. e.g. try:

```
  divu #0,d0
```

```
sorry about that! Division by zero causes a crash. Oh, well. Very few things
will cause a crash, unless you poke something into memory. e.g. consider:

  sub.l #64,a7
  move.l #1,62(a7)

move.w would have been ok, .l 62(a7) pokes part of you #1 outside your
64 byte play area, so when you add.l #64,a7 and RTS, you'll crash, since
you mucked up the return address there.
```

1.157 less27

```
Lesson 27 - Include Files

Now load  20.i, but don't assemble.

Now you cannot assemble this file without error as it stands; it is an
"include" file, and that is why it has the suffix .i instead of the usual
.asm for assembler files, and is in the  Includes  drawer.

Now, load and assemble  Teaching/20.asm   which  be assembled.
This program is very like 15.asm, but it replaces several lines by an
INCLUDE pseudo op. What happens is this:

  - Tandem assembles as normally

  - But if it comes to an INCLUDE it looks in a special memory buffer for
    INCLUDE's, to see if it can find the file "20.i". If it can't, it
    loads "20.i" into the INCLUDE buffer.

  - If Tandem loads  20.i  into the includes memory buffer, it stays
    there - even if you change the sc, the include files already loaded (if
    any) do not disappear from memory, unless you flush the includes memory
    buffer.

After assembly, press the "View sc" button, and you'll see 20.i in the list
of things in memory.

After the include .i file is loaded, Tandem reads subsequent sc from the
include file, until it is exhaused, and then jumps back to your sc. Include
files can in turn call other include files, down to a nesting depth of 10.
In the Main window you will see the include file's mc lines listed, and you
can single step or breakpoint in them just like in normal sc. But you cannot
edit the sc of files in the Include buffer (obviously). Only very thoroughly
debugged sc should ever be made into .i include files!
```

1.158 less28

```
Lesson 28 - Making a Program that can Execute from the Workbench

To make a program executable from the workbench, you must begin the
execution of your program by getting a signal from the workbench to your
```

program. You cannot single step this, since Tandem operates from the CLI, and so does not send and receive signals to and from the workbench. So when you write a program:

1. you first test whether it was running under the workbench or the CLI

2. if it is running from the workbench, you must receive a signal from the workbench.

3. then after your program finishes executing, you must reply to the signal from the workbench before you RTS.

I have included in Tandem a file IncAll.i which is a special case of an INCLUDE file. If you examined it, you would see it is full of MACRO's. But also, when Tandem assembles, if your program contains:

```
INCLUDE 'IncAll.i'
```

then Tandem will give you access to the values of thousands of constants, which are included in the standard Amiga INCLUDE files. There are lots and lots of Amiga INCLUDE files, and it takes lots of memory to assemble them all. But if you include IncAll.i, it is as if you have assembled them, in zero time! You will see all the Amiga .i files in the Includes and Autodocs manual, and you will no doubt find yourself referring to them many times, as a programmer. Also, if you have included IncAll.i, you don't need to put XREF's for ROM library calls, or an XREF for _AbsExecBase.

First, load and examine Teaching/21.asm, which illustrates the use of INCLUDE 'IncAll.i'.

Now, here is a program which does the message sending and receiving required from the workbench, if applicable: Teaching/22.asm

In order to explain the program, I will first give a summary of what it does:

* am I operating from workbench? If yes, get startup message
* open intuition.library. If can't, abort (quit bad)
* beep all screens
* am I operating from workbench? If yes, reply to startup message
* close intuition.library
* quit good

First, the program must decide if it is operating from the workbench or the CLI. Here is what it does:

```
 MOVE.L _AbsExecBase,A6 ;(note no XREF needed since IncAll.i)
 SUB.L A1,A1            ;the FindTask Autodoc says set A1=0 to find the
                        ;current task
 JSR _LVOFindTask(A6)   ;the Autodoc says this sets D0 to the task sought.
                        ;Since we set A1 to null, that will be the current
                        ;task.
```

The Amiga ROM Library calls always preserve all registers except D0,D1,A0, and A1. (It is a good idea to do the same with your subroutines, for consistency). The result (if any) of a call is returned in D0, and in rare instances a secondary result is returned in D1.

The Task Address actually returned in D0 is in fact the pointer to a "Process Structure", since your program is a process. A task that can use diskfiles is called a "Process". All tasks running from the workbench are processes. The CLI is also a process, and programs running from the CLI are not separate tasks but share the process structure of the CLI they run from.

What is a "process structure"? Well, the Amiga operating system maintains in memory lots and lots of little blocks of memory which it calls "structures". The operating system is always creating and disposing of such structures at a great rate of knots. Most of the structures you never need know about. But every now and then, one of them is important. For example, when you open a library, the Amiga looks through a list of libraries in memory, which are recorded in a list of items called "nodes" which are all linked together. If it finds that library already open, it will note in that library's structure that you have "opened" it, in a count of tasks using that library. If it is not open, the Amiga will load it from disk, and create a task structure for that library, part of which will actually point to where the library is in memory. And the library's structure will be in turn pointed to by a new node in the list of open libraries. If you "close" the library, the number of users of the library will be decremented. If there are then no users, the library can be deleted from its node in the list of open libraries, and the memory used by the library freed for other use.

This all happens in the background, and you don't need to think about it. But one example of a structure you do need to know about, is your program's own process structure, a node in the list of processes and tasks currently in memory. The process structure is not part of your program, nor do you create it or directly change it. The structure is created when someone clicks your program's tool icon. The Amiga loads your program into memory. Then, it creates a process structure for your program. Then, every now and then, the Amiga runs your program for a while if its turn comes up in the list of processes and tasks in memory. In that list will be a node pointing to your program's process structure. And in that process structure in turn will be a pointer to your mc itself. There will also be pointers to all sorts of other data, most of which is of no conscious interest to you.

In the Include files in the Includes and Autodocs manual, you will find an include file:

```
dos/dosextens.i
```

in which is the Process structure. And in the Process structure is:

```
BPTR pr_CLI     * pointer to command line interface
```

If pr_CLI is non-zero, your program is run from the CLI. If it is null, the program is not run from the CLI, therefore it is run from the workbench. So, you have to look in pr_CLI of your Process structure. To do that, you put D0 into an A register, so you can use an addressing mode, and look at pr_CLI, like this:

```
pr_CLI: EQU 172

MOVE.L D0,A2
TST.L pr_CLI(A2)  ;EQ if from workbench; NE if from CLI
BNE.S Cli         ;go if CLI
```

If you were to dissect the dos/dosextens.i include file, you would find that what BPTR pr_CLI actually does (by the BPTR MACRO), is to let

```
   pr_CLI: EQU 172
```

But you don't need to work all that out. Once have included IncAll.i, all you need to know is that you want pr_CLI. You don't need to know that pr_CLI = 172, since IncAll.i tells the assembler that for you.

Okay, that's all very well, but the big questions are these: How did I know all the stuff above? Why did I look in the process structure? Why did I focus on pr_CLI of all things? &c. &c. All this knowledge takes lots of time to acquire. You begin as you have done until now, and you read everything relevent you can, making notes as you go. People in Amiga user groups can help, too, if you can understand them. The Amiga ROM Kernal Manual is something you will soon be ready to start reading, and trying to understand. It is a bit tricky, since the examples are in C rather than assembler. But you soon see in a general way what the C examples are about, without knowing too much about C. There are plenty of introdutory books about C if you want to get a smattering of it. For example, the C way of writing pr_CLI in the Process structure is something like:

```
  Process->Pr_CLI     (think of Process as a DIM, and Pr_CLI as an element)
```

If I find that pr_CLI(A2) is non-zero, i.e. I started from CLI (as you will always find if you single step, since Tandem operates from the CLI, and your program operates from Tandem), then the program jumps to Cli. Else, it waits for a message:

```
 LEA pr_MsgPort(A2),A0 ;wait for workbench startup message to appear
 JSR _LVOWaitPort(A6)
```

The autodoc says, to wait for a message, put the address of a "message port" in A0. Now your task structure has an item:

```
 pr_MsgPort
```

which can receive messages for your process, and send messages from it. In this way, all the tasks in memory can communicate with each other. All sorts of messages are in fact flying back and forth all the time. When the workbench creates a process for your program, it sends a message to that process which means "you may begin". If you start without waiting for that message, the system will crash. So, you wait until a message appears at your message port. The WaitPort will cause your program to sleep until a message appears. Then, your program will wake up, and the next step is to grab the message, which you do by:

```
 LEA pr_MsgPort(A2),A0 ;get workbench startup message
 JSR _LVOGetMsg(A6)
 MOVE.L D0,message     ;remember the message, for replying
```

the Autodoc for GetMsg says to put the message port's address in A0, so that's why the first line puts it there. (Remember, library calls wipe out A0,A1,D0 and D1, so you have to fill in A0 again after WaitPort). Now the message you get will not be very thrilling. All it is, is a pointer to a message structure. This structure will stay in memory to look at, if you

want. For example, if the icon that started your program was a project icon, then your program will be that icon's tool, and the message will contain a pointer to the icon's name, i.e. a file for your program to work on. But if the icon was a tool, then the message will be no direct use.

Nevertheless, you must remember the message, ready for when your program closes down. That is why it is put in a DS.L called "message".

Now, your program does its thing. Finally, it is ready to return. The last thing you normally do is to close libraries. And even after that, you need to reply to the startup message, if you started from the Workbench. Here is the relevent part of the program:

```
 tst.w workbench       ;go if CLI
 beq.s Cliback
 jsr _LVOForbid(a6)    ;stop multi-tasking until RTS
 move.l message,a1     ;reply to message
 jsr _LVOReplyMsg(a6)
Cliback:
```

First, you bypass to Cliback if you are operating from the CLI. If not, you first execute a ROM call called:

```
 _LVOForbid
```

which stops the Amiga from multi-tasking, until you reverse it or your task finishes. When multi-tasking stops, all the clever things the Amiga does also stop, so you should almost never use it, and if you do, only for a few millionths of a second (as here).

After Forbid, you then put the message in A1, and call ReplyMsg. Now you don't need to know where to send the reply. The message structure contains a pointer to the port where the reply is to be sent. Your reply is merely to send the message back. The port that gets it is the port that sent it, i.e. the workbench. And the workbench knows when it receives it, that your program has finished. So, it removes your program from its list of tasks, and frees up the memory where it and the process structure resided. Now, supposing by a coincidence, that just after you sent the message, the task switching switched off your program before you finished closing the libraries, and switched to the workbench, which deleted your program. That would be terrible, because even if no task were using the libraries you opened, the system would think they were still in use, and so they would remain in memory wasting memory. That's why I put a Forbid just before sending your message. When the workbench removes your program's process, it also checks to see if Forbid is in effect from it, and if so, cancels the Forbid. You should never use Forbid in any other way than this until you are a very experienced programmer (and rarely if ever then).

Testing it Out

Having said all that, let's try it. Do the following:

1. Assemble Teaching/22.asm, and save the mc to RAM:Beep

2. Press Left Amiga/M to get back to the workbench, and open a Shell. From there, issue the command:

```
  RAM:Beep
```

which should beep all screens.

3. Now, close the Shell, and click the RAM: icon. Use the workbench "Window" menu and select "Show" with the right mouse button, to create an icon for RAM:Beep. Click the Beep icon, and watch the screen beep. Success!

1.159 less29

```
                Lesson 29 - Front.i
```

Now it is time for me to explain the Front.i include that comes with Tandem. So, load into sc the file:

Includes/Front.i (not to be assembled direct - it's an INCLUDE file)

There is a section of this guide about the Front.i program. In particular, you should now peruse and understand as well as you can:

```
                Front.i
                     You will not yet understand the bits about
                                 CD'ing to the PROGDIR:
```

Here is some extra explanation, to help you understand:

1. Front.i begins with TST.L (A7). Now since this is the return address for your program, it will always be NE. So why do it? The reason is this: if Tandem is debugging your program, it will change the NE to EQ. Front.i puts 0 or -1 in a DS based on this EQ or NE, so that way your program can tell if it is being debugged, or run normally.

2. Next, Front.i creates a 1024 byte memory block in the stack, which it fills with data, by treating it as an xxp_tandm structure. You will find the file:

   Tandem/Support/tanlib.i

   which has the xxp_tandm structure and its sub-structures. tanlib.i is also included in IncAll.i, along with the Amiga OS3.1 .i files, so all the xxp_ constants in tanlib.i are also available to your program.

   Front.i keeps A4 pointing to the 1024 byte xxp_tandm structure in stack. So for example, when it puts D7.W in xxp_tand, it uses A4 like this:

   ```
   MOVE.W D7,xxp_tand(A4)
   ```

   When Front.i calls your Program, it will have A4 pointing to the 1024 byte xxp_tndm structure. You can use the first 512 bytes as a scratch area if you like, and if you make use of tandem.library the other 512 bytes are also useful (else, leave them alone, and just use the first 512 bytes). You can of course simply ignore the 1024 bytes, and use A4 for something else.

3. When someone issues a CLI command, it can be followed by command parameters. e.g.:

```
List DF0:               (List is a command, df0: is its parameter)
```

   If someone starts your program from the CLI, and appends parameters, then the CLI sets D0 and A0 at startup as follows:

   A0 points to the parameters string as input i.e. 'DF0:',$0A
   D0 gives the string length. There is no 0 delimiter; and the string always ends with an LF ($0A). So if there is no parameter, D0 will be 1, and A0 will point to an LF.

   When you start Tandem, Tandem does not use parameters. However you can if you wish append typical parameters which the program which you are debugging might want to use. When Tandem assembles your program, and builds your startup stack, it also places the the A0 and D0 that the CLI gave to Tandem in your program's A0 and D0. A0 and D0 would have no meaning if your program is running from the workbench.

   Front0.i creates xxp_A0D0, and puts your program's initial A0 in in xxp_A0D0, and D0 in xxp_A0D0+4. If your program is running from the CLI these will be the parameter address and length. If from the workbench, they will be meaningless.

   (The workbench equivalent of parameters is to make your program the tool of a project icon. The name of the project file can then be retrieved from the startup command - see the RKM "Workbench" chapter).

2. dos.library is opened first, since setting up input and output requires routines in the dos.library. I have not yet covered files as a topic, but at this stage, I will mention the following:

   * when dos.library opens a file, it returns a value in D0 called a "file handle".
   * if you write to a file, or read from a file, or close a file, you must have the file handle in D1.

3. If your program runs from the CLI, you can write to the CLI window as if it were a file(!). All you need is a handle. For this reason, dos.library has a routine which tells you the output file handle of the process's CLI window. To get the handle:

```
move.l dosbase,a6
jsr _LVOOutput(a6)
```

   and dos.library puts the CLI output file handle in D0. Front0.i saves D0 in output: DS.L 1 whence your program can use it.

4. The same thing applies with _LVOInput which puts the CLI input file handle in D0, when Front0.i puts it in input: DS.L 1

5. But if you are in the workbench, there is no automatic window for input and output. So, Front0.i creates one, by:

```
move.l dosbase,a6
move.l #.conn,d1
```

```
   move.l #MODE_NEWFILE,d2
   jsr _LVOOpen(a6)
   move.l d0,input
   move.l d0,output

 .conn: dc.b 'CON:20/20/400/100/Console',0
```

Some of the labels in Front.i are called "local" labels: labels can be "local" or "global". Most labels are global - they start with a letter, and consist of a string of letters and numbers; the underscore _ counts as a letter. But if a label starts with a . it is a local label. Now all global labels must be unique within a program (except for the label of a SET pseudo op). But local labels must be unique between 2 global labels. Between any 2 global labels, there can be a series of local labels, which are defined only in the context between those global labels, and whose values vanish outside them. Front.i uses all local labels, except TLColdstart. This is useful, because an include file should only use labels which are "passed" to the program which includes them, otherwise it is a nuisance.

The above lines actually open the console for input and output, as if it was a diskfile! The autodoc for Open says to point D1 to filename, and to put the "access mode" in D2. The appropriate access mode is MODE_NEWFILE, which is one of several modes of opening diskfiles. See the AmigaDOS manual for details. Now the string at .conn is a filename, though it doesn't look like it. See again under AmigaDOS consoles. You can, for example, open consoles from the CLI if you want. When you open a file, Open returns its file handle, which can be used for input and output both.

Isn't it amazing that a window (or a printer, or memory, or a CD, or lots of things) can mimic a diskfile? For each thing, the Amiga has a piece of software called a "device". For example, for the printer, there is a device called "printer.device". A device basically says: "treat me like a diskdrive". dos.library, whenever it reads or writes, creates little structures called "packets". The Amiga keeps sending zillions of packets between devices, just like it keeps sending message structures between tasks. The structure for a packet is just the same, regardless of the nature of the sending and receiving devices. And the packets are sent out "asynchronously" - that is, the sender sends them, and forgets them, and goes on to something else. Somewhere there is a task moving packets around. Just as you don't often need to think about individual messages between tasks, so still less do you need to think about individual packets. You can if you like do things at the packet level, but it is far simpler to do them at the File level.

6. Your program can use xxp_memky to reserve memory blocks, like this:

```
   move.l intbase,a6
   lea memkey,a0
   move.l #xxx,d0            ;replace xxx by the size of the block
   moveq #MEMF_PUBLIC,d1     ;see Amiga ROM Kernal manual for memory types
   jsr _LVOAllocRemember(a6)
```

and D0 returns the address of your block. The real power of this is that at the end, a single call to FreeRemember (which Front.i does, saving you the trouble) frees up all the blocks at once. intuition.library does this trick by creating a structure with nodes for each allocation using xxp_memk.

oOo

Incidentally, the Amiga hardware is continually producing events called "interrupts" many times a second. Each interrupt causes the following:

1. all your registers push onto your stack.

2. the 680x0 goes to a new mode of operation, called "supervisor mode". A7 changes to a new value called the "supervisor stack". It then jumps to an address dictated by the type of interrupt that happened. For example, if you reboot the computer, it is actually an interrupt. Similarly, all the devices can cause interrupts. Your software can too, for example if you divide by zero.

3. the Amiga's operating system intercepts all interrupts, and exercises a routine called an "interrupt service routine". This ends with an opcode RTE, which switches A7 back to your stack, pops all your registers back, and your program restarts as if nothing happened.

Interrupts are "transparent" to your program. It continues as if the interrupts weren't happening. Although the interrupts are absolutely vital to the functioning of the computer - even if interrupts were disabled for a few milliseconds, the system would fail spectacularly - yet they only take up a miniscule part of the computer's time.

In the 680x0, the interrupts are sometimes called "exceptions" or "traps". If you see references to exceptions, or traps, it is interrupts that are being spoken of. Many of the ROM calls you make generate interrupts. You will probably never need to use interrupts directly, or understand much about them.

Incidentally, if a machine code program is known to jump to a certain address, you can intercept it, by putting a JMP there. For example, some interrupts jump to addresses near $0. The Amiga puts a JMP at each such. JMP's that intercept other taks, do something, and then jump back to the task, are called "vectors". You can do all sorts of jazzy things by putting vectors into the Amiga operating system. For example, you will see in the AmigaDOS manual something about SetPatch, which is a type of vector. Well written applications programs should never use interrupts or vectors.

You will also see here and there in the Amiga ROM Kernal manuals references to "hooks", which are vectors into the Amiga's system. These are advanced usage, and few programmers ever need to use hooks. In general, you should NEVER put anything in memory, without using AllocRemember (or some equivalent) to reserve it to your use. And you should never poke anywhere else, but use the proper ROM library calls, which do it safely.

1.160 less30

Lesson 30 - Using Front.i

The program Teaching/23.asm shows the correct usage of Front0.i Load it, and step it through. It does the following:

* sends a message to output

```
   * gets a message from input

1. Save it as RAM:Temp

2. Now, open a new CLI, and issue the command

               RAM:temp

   You will see the message 'Hello, world'

3. Then a cursor will appear. Input a message. RAM:Temp will receive it, but
   do nothing with it, but simply close down.

4. Now, go to the workbench, and use the right mouse button window menu item
   "Show" to make an icon for RAM:Temp. (You could also use the IconEdit
   tool to create an icon, which would be a file named  RAM:Temp.info).

5. Now, after opening the RAM: (RAM Disk) window, and making RAM:Temp
   visible, you will hopefully see an icon for Temp, which click.

6. If all goes well, a window labelled "Monitor" will appear, with the
   message and cursor as per the CLI. But when your program finishes, the
   monitor disappears, whereas when you ran it from the CLI the CLI remains
   after your program finishes.

There are more Teaching/x.asm files, where x=24+. You should load and
inspect and step these - they have ample comments which hopefully will prove
instructive. By the time you've done all that, you'll be able to use and
understand the Amiga ROM Kernal manuals (you should read the relevent parts
of these as you work through the Teaching files).

Perhaps I'll leave it at that. Go out and write something stunning. Welcome
to the fellowship of Amiga assembly language programmers.

An Ancient Irish Programmer's Blessing

May your pushes match your pops
May all your .W's and .L's be to even addresses
May you never violate the Amiga protocols
May you remember to de-allocate everything you allocate
May you find your way through the includes and autodocs
May you never omit extensions
May you never put CMP addresses the wrong way round
May your divisor never be zero
May you get control of the blitter when you want it
May you never have to debug a printer driver
May you never leave off a #
May you never forget to specify chip ram
May all your bugs appear when you are debugging          Ken Shillito
May you always save your sc before a crash               October, 1997
```

1.161 front

Optional Supplement - Front.i and tandem.library

```
Here is some information about a set of files:

   Tandem/Includes/Front.i          Tandem/Support/tanlib.i
   Tandem/Includes/Tandem.i         Tandem/Support/Tandem.FD

   Tandem/tandem.library  (may optionally be put in LIBS:)

These are used to some extent in the
                Tandem Primer
                , and for
Tandem's internal workings.

This is full documentation, if you want to use them. However you can use all
aspects of Tandem without ever making use of the above files.


                Overview of Tandem Includes

                How to use Front.i

                How to use tandem.library
                I would suggest that you load and assemble Tandem/Teaching/21.asm  ↩
                   through to
Tandem/Teaching/72.asm, which will give you a general intuitive feel for
how tandem.library works, and what it does. You will then find it much easier
to read this segment of the guide.
```

1.162 fver

```
                Overview of The Tandem Includes


1. IncAll.i

IncAll.i is part of a system that includes the following files:

   Tandem/Support/incall.consts   <- all the Amiga .i files, pre-assembled
                                     + tandem/support/tanlib.i

   Tandem/Support/ssXref.consts   <- all the Amiga FD files, pre-assembled
                                     + tandem/support/tandem.FD

   Tandem/Includes/IncAll.i       <- the Amiga and tandem.library MACRO's

These four files mean that all the Amiga OS release 3.1 constants are
pre-assembled, and instantly available to your program! This is explained
in
                INCLUDE
                 under IncAll.i.

Also pre-assembled with the above is Tandem/Support/tanlib.i and
Tandem/Support/tandem.FD

The above are all transparent to the user. Basically, you can simply omit
all INCLUDE's of any tandem .i files, and simply INCLUDE 'IncAll.i'. This
```

will be as if you INCLUDE every Amiga .i file, except they are all instantly assembled!

2. Front.i

Front.i is a standard front-end for your programs. I suggest you always have the following in your programs, at the start:

```
        INCLUDE 'Front.i'
```

This does all the routine setting up (including INCLUDE 'IncAll.i') such as receiving a startup message, opening libraries, then calling your program, and finally closing libraries and replying to the startup message.

Front.i also detects if your program is being debugged by Tandem, and if so, facilitates that (more on that below).

It is certainly not compulsory to use Front.i, and Tandem will work perfectly well on programs that do not INCLUDE it. Front.i is merely a convenient suggestion to save you work. If you never want to use Front.i then you need read this document no further. You will no doubt have your own "front end" equivalent to Front.i

If youuo do use Front.i, then you must have a line "strings: DC.B 0", followed by 0 or more strings, and the entry point to your program muast be labelled "Program". Here is a minimum use of Front.i:

```
  * a proram that includes Front.i

   INCLUDE 'Front.i'

  strings: DC.B 0
   DS.W 0

  Program:

   RTS
```

Obviously, you'd of course want to put something between Program and its RTS!

3. Front.i and tandem.library

In Tandem's PROGDIR: you will see the following file:

```
   tandem.library            (or, you may have moved it to LIBS:)
```

Tandem needs this for its internal workings. It is in fact an integrated set of files to support the type of user interface which Tandem gives you. If you use the

```
                Tandem Primer
                  to teach yourself assembler,
```

Tandem includes calls to tandem.library in its method of instruction.

Many users after learning with Tandem will want to use tandem.library in

their own programs. If so, any program that uses tandem.library calls can include the following:

```
   INCLUDE 'Front.i'
```

since as well as setting up a normal front end, it opens tandem.library

I should re-iterate that if you never want to use tandem.library, you need read this guide no further; you can simply use Tandem to assemble under your own system; most users will build up gradually for themselves something equivalent to tandem.library

4. Tandem.i

Tandem.i is of no direct use: it is, in fact, the source code for tandem.library. It also contains within itself Front.i.

Comments at the start of Tandem.i tell you how to assemble tandem.library from it, or how to make it into Front.i

Whenever I want to update or modify tandem.library, I in fact modify Tandem.i, and then from it re-create updated versions of Front.i and tandem.library. Thus, Tandem.i is the basic document from which the others spring.

You can, if you wish, modify tandem.i and create your own version of Front.i and tandem.library: but please:

IF YOU MAKE YOUR OWN MODIFIED VERSION OF TANDEM.LIBRARY - PLEASE GIVE IT A DIFFERENT NAME!!!!!!

(The same applies to Front.i).

Of course, I retain the right and obligation to bring out updates of tandem.library. I have numbered the version with this release, version 37 of tandem.library. Front.i requires version 37+ of the Amiga OS libraries, i.e. release 2.04+ of the Amiga operating system.

You can if you wish (for development purposes) include Tandem.i instead of Front.i in your program. This means that, when you make calls to tandem.library, Tandem can set breakpoints and step through the actual instructions of the library calls. But, when making machine code for saving, you should of course use Front.i which makes much more compact machine code.

1.163 frt0 Front.i - An Assembly Language Frontend

How to Use Front.i

An experienced programmer can read through Front.i easily enough, as it is a typical Amiga assembly language "front end". To use Front.i you must:

1. put the statement:

```
        INCLUDE 'Front.i'
```

   at the start of your program.

2. include a line:

```
        strings  DC.B  0
```

   followed by 0 or more null delimited strings.

3. label the entry point of your program

```
        Program
```

e.g. here is a minimal use of Front.i:

```
       * a program to do nothing

        include 'Front.i'

       strings: dc.b 0
        ds.w 0

       Program:
        rts
```

Summary of what Front.i does

(You should put Front.i in the "Jotter" window, and arrange the jotter and AmigaGuide windows so you can look at Front.i & the summary below simultaneously.)

1. Creates the following label for internal purposes:

```
       xxp_what EQU 1
```

   and the label:

```
       TLColdstart
```

   at relative address $0000 being the entry point of the program.

2. INCLUDE's IncAll.i
   IncAll.i, as well as defining all the Amiga OS3.1 .i and FD labels, includes a lot of labels & MACRO's starting with:

```
       xxp_.....  (constants)              e.g.  xxp_sysb
       TL....     (MACRO's)                e.g   TLwindow
       _LVOTL...  (tandem.library XREF's)  e.g.  _LVOTLWindow
```

   These are documented in:

```
   Tandem/Support/tanlib.i     (the xxp_... constants and TL... MACRO's)
   Tandem/Support/tandem.FD    (the _LVOTL... constants)
```

3. Creates a 1024 byte area in stack, an instance of an xxp_tndm structure. The first 512 bytes of this are a general purpose scratchpad, while tandem.library and your program use the things in the second 512 bytes, referred to as xxp_ labels offset to A4. The xxp_tndm structure and its subsidiary structures are documented in Tandem/Support/tanlib.i, and all the xxp_ offsets are made available by INCLUDE 'IncAll.i'. Front.i always keeps A4 pointing to the 1024 byte xxp_tndm structure.

4. Sets xxp_tand(a4)=-1 if debugging under Tandem, else sets xxp_tand(a4)=0

5. Caches initial A0 in xxp_A0D0+0(a4) }These are the argument string, if
   Caches initial D0 in xxp_A0D0+4(a4) }operating under CLI

6. Caches _AbsExecBase in xxp_sysb(a4)

7. Sets xxp_bnch(a4)=0 if your program is operating under the CLI, whereupon xxp_A0D0 has meaning (see 5 above).

8. If your program is operating under the workbench, receives the system startup message and puts it in xxp_bnch(a4)

9. Initialises things that might be used for tandem.library

10. Tries to open dos.library, & puts its handle in xxp_dosb(a4).
    If it cannot open any version of dos.library it aborts (unlikely).

11. If in CLI, puts the _LVOInput handle in xxp_iput(a4)
    If in CLI, puts the _LVOOutput handle in xxp_oput(a4)

    If in workbench, opens a CLI-like "Console" window, and puts its handle in xxp_iput(a4) and xxp_oput(a4). It aborts if it cannot open the Console window (unlikely).

12. Tries to open the following libraries:

```
intuition.library     puts handle in xxp_intb(a4)
graphics.library      puts handle in xxp_gfxb(a4)
asl.library           puts handle in xxp_aslb(a4)
gadtools.library      puts handle in xxp_gadb(a4)
tandem.library        puts handle in xxp_tanb(a4)
```

    The minimum version of tandem.library is xxp_tver, currently SET to 37.
    The minimum version of the others, and dos.library, is xxp_lver, which
        is currently SET to 37.
    Front.i aborts if the minimum versions cannot be opened.

13. BSR's Program

    You must label the entry point of your program as Program. Your program then does its thing until it RTS's back to Front.i

    When your program is called:

```
        A0,D0  are set to the xxp_A0D0 values
        A4     points to the aforesaid 1024 byte xxp_tndm structure in stack
        A6     contains xxp_sysb(a4)   (i.e. exec.library base)

    The 4 bytes above (A7) are filled with the 4 bytes above (A7) at
    TLColdstart, reduced by the stack usage so far. If under the CLI, this
    will be the stack size available to Program.

14. Program can, if you wish, make _LVO calls to tandem.library

    All calls to tandem.library must have A4 pointing to the xxp_tndm
    structure in stack, which Tandem.i passed to Program.

15. CD'ing to your PROGDIR:

    See
                CD'ing to the PROGDIR:
                 and read it carefully.

    Front.i zeroises xxp_cdir. If Program calls TLProgdir, it puts the old
    CD in xxp_cdir(a4).

16. After your program returns:

    (a) Front.i sees if Program called TLProgdir. If yes, it CD's
        back to the original CD cached in xxp_cdir(a4).

    (b) If Program called TLbad, it waits on the CLI/Monitor for
        the user to acknowledge.

    (c) If Program called TLPublic or TLChip, it calls _LVOFreeRemember.

    (d) If Program opened a private screen with TLscreen, it closes the
        screen.

17. If Program has called TLWindow, it calls TLWclose to free all resources
    used by tandem.library. (i.e. tandem.library does resource tracking -
    Program does not need to close down anything created by tandem.library
    calls).

18. Closes the libraries Front.i opened.

19. Replies to the workbench startup message if applicable.

20. Returns with D0=0, i.e. ok. But if Program called TLbad, it returns
    with D0=-1 (i.e. an error condition if running under CLI).

This concludes the discussion of Front.i. The rest of this document
pertains to tandem.library, so if you don't wish to use tandem.library you
needn't read it.
```

1.164 fmac tandem.library - an assembly language GUI

```
                How to use tandem.library
```

```
Below are the docs for tandem.library. Start with the first column, and
progress to the second column.

Elementary Routines          GUI Routines

                about Tandem.i

                calling TLWindow
                 <- Important!!

                about tandem.library

                window routines

                string moving etc routines

                i/o routines

                type conversion routines

                menu routines

                memory allocation routines

                rendering routines

                CLI/Monitor routines

                requester routines

                file routines

                primitive routines
```

1.165 (pram) About Tandem.i

```
                About Tandem.i

All the data initialised by Front.i are in a standard block of 1024 bytes,
being 512 bytes for scratch, and 512 bytes for all data storage, including
xxp_WSuite, an instance of a xxp_wsui structure, which maintains a suite of
up to 10 windows for you.

In Tandem/Support/tanlib.i, you will see that the xxp_wsui structure has 12
xxp_wsuw stuctures, one for each window. If tandem.library opens any of
windows 0-9 (by a call to TLWindow), its xxp_wsuw structure is intialised.
window 10 is reserved for requesters, and window 11 for help.

All calls to tandem.library must have A4 pointing to the xxp_tndm structure
in stack which was set up by Front.i before it called Program. It is a good
idea to allow more than the minimum 4096 bytes in stack if calling
tandem.library. See
                Stack Usage
                .
```

```
The 512 byte scratchpad at (A4) is labelled xxp_buff(A4). In the
amiga.library docs, this is sometimes called "buff" or "buffer"
for short.

e.g. the MACRO reference    TLstrbuf #5   would copy string 5 to buff -
i.e. it will copy string 5 to (A4).
```

1.166 (auto) About tandem.library

```
About tandem.library - Introduction

All routines require A4 to point to the 1024-byte structure in stack that
Front.i creates for you. Front.i calls Program with A4 pointing there for
you. This is an instance of a xxp_tndm structure. The xxp_tndm structure,
and its subsidiary structures, are detailed in Tandem/Support/tanlib.i

I have included Tandem/Includes/tandem.i (being the sc of tandem.library),
so you can see how tandem.library does things. I encourage people to
experiment and tinker with these routines (but if you make a new version
of tandem.library, please rename it).

All tandem.library routines change only those registers that return results;
all other registers are always returned unchanged.

Every routine below has a MACRO  which loads its registers and calls it.
These MACROs save all registers except those in which results are obtained.
The MACRO details are listed as MACRO: in each autodoc. The MACROs use
MOVE.L to put the parameters into the the regs, so put # before values, else
an address is assumed. e.g.: if \1 is #5 (for D0), it becomes  MOVE.L #5,D0

So e.g. if D0 already holds the value wanted for \1, put D0 for \1, and
the MACRO creates MOVE.L D0,D0 which works.

Routines that return a NE(good)/EQ(bad) (=Boolean) result in D0 have D0
saved in their MACRO's, which set EQ/NE on D0 as returned, before restoring
D0 as it was when called. But the library call does not of course save D0.
(Amiga guidelines say libraries should not return results in CCR).

Caution: registers are filled in increasing order, so you cannot
put this for example:

    TLReqchoose d1,d0

since this would result in:

    ....
    move.l d1,d0
    move.l d0,d1
    ....

but TLReqchoose d2,d1 would be ok, since it results in:

    ....
    move.l d2,d0
    move.l d1,d1
```

```
    ....

which is ok.

i.e. if you use registers as parameters (as above, where \1=d1 and \2=d0)
then you shouldn't use Dn for \m, if n<m, or the registers will get mucked
up. Make sure you understand this.

Now, I will give autodocs for each MACRO & its corresponding library call.

n.b. If you work through the sc files in Tandem/Teaching, you will
see many detailed examples of how to use TL calls effectively. The autodocs
give sample programs in Tandem/Teaching for each tandem.library call.
```

1.167 stri

```
                String Routines

tandem.library has two 2 routines for finding & moving strings by number:

TLstra0         ;point A0 to string
                TLStra0
                TLstrbuf        ;tfr string to buffer
                TLStrbuf
                Strings

All programs which call tandem.library INCLUDE Front.i at the start. And
programs which INCLUDE Front.i must have the statement:

            strings: DC.B 0

followed by 0 or more null-delimited strings. In all the TL routines, where
a string number is mentioned, it is the number of the string (1+) in the
DC.B's after strings. e.g. a set of strings:

            strings: DC.B 0
             DC.B 'My Project - version 0.0',0     ;string 1
             DC.B 'Please click to acknowledge',0  ;string 2
             DC.B 'copyright Otillihs Nek',0       ;string 3
             DS.W 0

Then, to use say string 3 above, you will typically place it in xxp_buff(A4):

            TLstrbuf #3

and then perhaps send it to the CLI/Monitor with:

            TLoutput            (or combine the steps with TLoutstr #3)

You will see many uses of the xxp_buff(A4). You can use the xxp_buff as a
general purpose scratchpad, as well as for TL calls.


Dynamic Strings   (Advanced Usage)
```

```
The table of strings labelled "strings" is meant for you to put all your
Program's string constants in. But is it possible for tandem.library to use
string variables in its string routines? Yes, here's how.

(a) your string variable(s) must be in a block with an initial 0 byte,
    followed by null delimited strings referred to by number 1+
    (they could be in stack, for example).
(b) tandem.library keeps xxp_strg(A4) pointed to #strings
(c) to use the strings in your block, cache xxp_strg, point xxp_strg
    to your strings, use them for whichever tandem.library calls, then
    restore the value in xxp_strg.

Like this:

    ; (sp) has a set of string variables, with a 0 at their start.
     move.l xxp_strg(a4),a0     ;cache xxp_strg
     move.l a7,xxp_strg(a4)     ;point to my string variables
     TLoutstr #3                ;send my 3rd string to the CLI/Monitor
     move.l a0,xxp_strg(a4)     ;restore xxp_strg

[instead of caching xxp_strg you could simply end with
move.l #strings,xxp_strg(a4).]
```

1.168 stbf

```
                D0 = TLStrbuf(D0,A4)

Puts string D0 in buffer, sets d0 to bytes transferred

See also
                Strings

                TLOutstr
                Call:  A4=1024 byte buffer created by Front.i
       D0=strings number

Back:  D0=string len in characters (0+, excluding the null delimiter)

Caution: The string length must not exceed 511 characters

MACRO: TLstrbuf    \1=string num; saves ALL registers (even D0)

Example:    [a4 set by Front.i]        or        [a4 set by Front.i]
            move.l xxp_tanb(a4),a6               TLstrbuf #5
            moveq #5,d0
            jsr _LVOTLStrbuf(a6)

Sample program:   Tandem/Teaching/23.asm
```

1.169 sta0

```
               A0 = TLStra0(D0,A4)

Points A0 to string D0

See also
               Strings
               Call:  A4=as set by Front.i
       D0=the string number to which A0 is to be pointed

Back:  A0 points to string D0

MACRO: TLstra0     \1=stringnum; result in A0

Example:    [a4 set by Front.i]       or        [a4 set by Front.i]
            move.l xxp_tanb(a4),a6               TLstra0 #5
            moveq #5,d0
            jsr _LVOTLStra0(a6)
```

1.170 type

```
               Type Conversion Routines

tandem.library has 4 type conversion routines:

TLaschex       ;convert ASCII to hex
               TLAschex
               TLhexasc        ;convert hex to ASCII decimal
               TLHexasc
               TLhexasc16      ;convert hex to ASCII hex
               TLHexasc16
               TLfloat         ;convert ASCII to float
               TLFloat
               TLfloat does not require a maths coprocessor (or 68040+) to run,  ↩
                  but its
output would presumably always be used to FMOVE.P to an FP register.
```

1.171 ashx

```
D0 = TLAschex(A0,A4)

ASCII to hex conversion (unsigned double integer).
Not highly optimised for small values; result is modulo $10000000.

Call:  A4 as set by Front.i
       A0=string addr (if no number there, A0 returns unchanged, D0=0)

Back:  D0=value
       A0 points to 1st chr outside range '0'-'9'

MACRO: TLaschex       \1=string address    does TST.L D0 on return
                      saves all except a0,d0; a0 points to delimiter
```

```
Example:    [A4 as set by Front.i]    or      [A4 as set by Front.i]
            move.l xxp_tanb(a4),a6            TLinput
            jsr _LVOTLInput(a6)               TLaschex a4
            move.l a4,a0                      move.l d0,hxnum
            jsr _LVOTLAschex(a6)
            move.l d0,hxnum

Sample program:  Tandem/Teaching/24.asm
```

1.172 hxas

```
() = TLHexasc(D0,A0,A4)

Hex to ASCII conversion, decimal (unsigned double integer without leading
0's). Not highly optimised for small values.

Call:  A4 as set by Front.i
       D0=the value to be output
       A0=the address to put the ASCII to

Back:  A0 points past last character ouput

MACRO: TLhexasc      \1=value to output  \2=address to put to

Example:    [A4 as set by Front.i]      or      [A4 as set by Front.i]
            move.l xxp_tanb(a4),a6              TLhexasc hxnum,a4
            move.l hxnum,d0                     clr.b (a0)
            move.l a4,a0                        TLoutput
            jsr _LVOHexasc(a6)
            clr.b (a0)
            jsr _LVOOutput(a6)

Sample program:  Tandem/Teaching/24.asm
```

1.173 ha16

```
() = TLHexasc16(D0,D1,A0,A4)

Hex to ASCII conversion, hexadecimal, with specified number of digits

Call:  A4 as set by Front.i
       D0=the value to be output
       D1=the number of digits to be output(1 to 8), or 0 to left justify.
          e.g. if D0=$000F3210, and D1=0, 5 digits ('F3210') will be output.
       A0=the address to put the ASCII to

Back:  A0 points past last character ouput

MACRO: TLhexasc16  \1=value to output  \2=digits  \2=address to put to

Example:    [A4 as set by Front.i]      or      [A4 as set by Front.i]
```

```
            move.l xxp_tanb(a4),a6              TLhexasc hxnum,#8,a4
            move.l hxnum,d0                     clr.b (a0)
            moveq #8,d1                         TLoutput
            move.l a4,a0
            jsr _LVOHexasc16(a6)
            clr.b (a0)
            jsr _LVOOutput(a6)

Sample program:  Tandem/Teaching/24.asm
```

1.174 flot

```
               D0 = TLFloat(A4)

Converts a decimal string into a floating point 68881/2 .P format
The number is normalised as it is converted.

For the rules for input strings, see
               DC
               .

Call:  A4 as set by Front.i
       A0 points to input string
       A1 points to 12 byte output buffer

Back:  D0=error code:  0=ok  1=no digits in mantissa  2=bad exponent
                       3=no digits after E/e
       If D0=0, can  FMOVE.P (A0),FPn
       If result is zero, 3(A1)=0
       A0 points to the delimiter (the 1st chr that does not logically
           belong to the number, e.g. a ',' a $00 or a space).

MACRO: TLfloat \1,\2   returns EQ if bad; returns with \2 still in A1,
       A0,D0 as per "Back" above.

Example:    [A4 as set by Front.i]      or      [A4 as set by Front.i]
            move.l xxp_tanb(a4),a6              TLfloat #input,#output
            lea input,a0                        beq Bad
            lea output,a1                       FMOVE.P (A1),FP2
            jsr _LVOTLFloat(a6)
            tst.l d0
            bne Bad
            FMOVE.P (A1),FP2

Sample program:    Tandem/Teaching/45.asm
```

1.175 mmrt

```
               Memory Routines

tandem.library has 2 memory routines:
```

```
TLpublic        ;create public ram
                TLPublic
                TLchip          ;create chip ram
                TLChip
                These use _LVOAllocRemember, with xxp_memk as a key. Front.i  ↩
                   initialises
xxp_memk, and when your program returns, Front.i calls _LVOFreeRemember if
required.

You *must not* call _LVOFreeRemember for memory called by TLpublic or
TLchip. So if you want to temporarily allocate memory, set up your own
rememberKey.
```

1.176 publ

```
                D0 = TLPublic(D0,A4)

Creates public memory.

See also
                Memory Routines
                Call:  A4 as set by Front.i
       D0=bytes to create

Back:  D0=address where created

MACRO: TLpublic      \1=bytes to create   sets EQ if bad

Example:     [A4 as set by Front.i]      or      [A4 as set by Front.i]
             move.l xxp_tanb(a4),a6               TLpublic #10000
             move.l #10000,d0                     move.l d0,work
             jsr _LVOTLPublic(a6)                 beq Bad
             move.l d0,work
             beq Bad

Sample Program:   Tandem/Teaching/24.asm
```

1.177 chip

```
                D0 = TLChip(D0,A4)

Creates chip memory

See also
                Memory Routines
                Call:  A4 as set by Front.i
       D0=bytes to create

Back:  D0=address where created
       D0=0 if out of chip mem

MACRO: TLchip        \1=bytes to create    sets EQ if bad
```

```
Example:          as per
                TLPublic
                Sample program:   Tandem/Teaching/24.asm
```

1.178 clim

```
                tandem.library CLI/Monitor routines

tandem.library has the following CLI/Monitor routines:

TLinput         ;get from input stream
                TLInput
                TLoutput        ;put to output stream
                TLOutput
                TLoutstr        ;combines TLstrbuf,TLoutput
                TLoutstr
                TLerror         ;error report
                TLError
                TLbad           ;report error on closedown
                TLbad
                When Front.i starts up under the workbench, it opens up a CLI-like ↩
                    monitor.
So, whether you run under CLI or workbench, you can use the CLI window (or
monitor) for old-fashioned terminal-style interaction with the user, using
TLinput and TLoutput.

TLError is for calling if an error occurs. First, it sees if there is a dos
error, and if so, sends it to the CLI (these are mainly useful only to
power users). When tandem.library calls return with errors, tandem.library
puts an error code in xxp_errn(a4), and if you then call TLError, it puts
an error report in (a4) which you can send to the CLI or put up a requester.

TLbad allows you to close down in such a way that an acknowledgement of the
CLI/Monitor contents is required. Use this if your program fails
catastrophically, such as running out of memory.
```

1.179 inpt

```
D0 = TLInput(a4)

Input from input handle to buffer; null delimits - i.e. chops off $0A

Call:  A4 as set by Front.i

Back:  D0=value returned by _LVORead (stringlen, +1 for the $0A chopped)
       String in buff, null delimited, final $0A chopped off

MACRO: TLinput       no parameters      saves all regs, even D0.

Example:    [A4 as set by Front.i]      or     [A4 as set by Front.i]
            move.l xxp_tanb(a4),a6             TLinput
```

```
            jsr _LVOTLInput(a6)
```

1.180 otpt

```
                D0 = TLOutput(A4)

Outputs from buffer to output handle (temporarily appends $0A).

Call:  A4 as set by Front.i
       buff contains a null delimited string

Back:  D0=value returned by _LVOWrite (=stringlen, +1 for the $0A appended)
       tandem.library appends $0A to the string in buff while writing, but
       then removed it before returning.

MACRO: TLoutput      no parameters       saves all regs, even D0.

Example (sends string 20 to CLI/Monitor):

            [A4 as set by Front.i]     or      [A4 as set by Front.i]
            move.l xxp_tanb(a4),a6             TLstrbuf #20
            moveq #20,d0                       TLoutput
            jsr _LVOTLStrbuf(a6)
            jsr _LVOTLOutput(a6)

Sample program:   Tandem/Teaching/24.asm
See also:
                TLoutstr
```

1.181 otst

```
                D0 = TLoutstr  MACRO only

This combines
                TLstrbuf
                 and
                TLoutput
                .

Example:    TLstrbuf #20     can be replaced by     TLoutstr #20
            TLoutput
```

1.182 qerr

```
D0 = TLError(A4)

Reports on the error status of xxp_errn(a4). Any tandem.library routines
that return errors put an error code in xxp_errn(a4).

You can put up a requester with the result in buff showing, but if there
```

```
is no memory even for that, at least an error report has been sent to the
CLI/Monitor.

Call:  A4=as set by Tandem.i
       Error code in xxp_errn(a4), as set by a tandem.library call

Back:  1. Sends dos.library TLFault (if any) to CLI/Monitor.
       2. Puts meaning of xxp_errn(a4) in buff.
          e.g. if xxp_errn(a4)=1, puts "Out of public memory" in buff.
       3. If xxp_errn(a4)<>0, sends buff to CLI/Monitor.
       4. Returns dos.library error number from step 1 in D0. (0 if none).

MACRO: TLerror     no parameters     returns EQ if xxp_errn(a4)<>0 (=error)

Example:    [a4 set by Tandem.i]        or         [a4 set by Tandem.i]
            move.l xxp_tanb(a4),a6                 TLerror
            jsr _LVOTLError(a6)                    beq Bad
            tst.l xxp_errn(a4)
            bne Bad

tandem error codes
 0  Cancel selected
 1  Out of public memory
 2  Out of chip memory
 3  Can't open file for reading
 4  Can't open file for writing
 5  Can't read file
 6  Can't write file
 7  Can't lock public screen
 8  Font operation failed - Can't open diskfont.library
 9  Can't get screen vi for gadtools.library
10  Can't open font
11  Object won't fit in window
12  Can't open half-height font (super/sub script)
13  Can't make double width font - Can't create FONTS:Temporary
14  Can't make double width font - Can't open FONTS:Temporary
15  Can't make dble width font - Can't write to FONTS:Temporary
16  Can't make double width font - NewFontContents failed
17  Fon't operation failed - Can't lock FONTS:
18  Can't make double width font - Can't open Temporary.font
19  Can't make double width font - Can't write to Temporary.font
20  Can't make double width font - Can't re-open Temporary.font
21  Requester won't fit in screen/window
22  Needs intuition.library v. 39+ (Amiga OS release 3.0+)
23  Can't create a prefs dir
24  Can't create a prefs file
25  Can't open ILBM file
26  Not an IFF file
27  Not an ILBM file
28  Garbled ILBM contents
29  Unrecognised ILBM compression method
30  LayoutMenusA failed (unlikely)
31  Edit error - window too narrow
32  Edit error - window too shallow
33  Edit error - can't obey fixed offset
34  Edit error - can't attach font
35  Operation cancelled because window resized
```

```
36  Can't make screen rendering objects (out of memory)
37  Can't get screen DrawInfo (out of memory)
38  Can't make window scroller rendering object (out of mem)
39  Can't put up font selector (too many windows already open)
40  Can't open printer device
41  Error in sending characters to printer
```

1.183 badd

```
TLbad     (MACRO only)

TLbad is a MACRO only, not a subroutine. It sends string \1 to the output
stream, and sets xxp_ackn(a4)<>0, so that Front.i will ask for acknowledge
(string \1 would be an error report), and then returns bad (with D0=-1).

Example: string 22 is    DC.B 'Program aborted: out of memory',0 ;22

         TLpublic #20000
         bne.s Good
         TLbad #22              <- when this program exits, string 22 will
         bra Abort                 appear on the CLI/Monitor, along with
        Good:                      "please acknowledge".

Sample program:   Tandem/Teaching/28.asm   (see under Pr_bad)
```

1.184 flrt

```
                File Routines

tandem.library has these general purpose file routines:

TLopenread      ;open file for reading
                TLOpenread
                TLopenwrite     ;open file for writing
                TLOpenwrite
                TLreadfile      ;read from file
                TLReadfile
                TLwritefile     ;write to file
                TLWritefile
                TLclosefile     ;close file
                TLClosefile
                and also these special purpose file routines:

TLassdev        ;see if a device/assign exists
                TLAssdev
                TLprefdir       ;create prefs dir
                TLPrefdir
                TLpreffil       ;create prefs fil
                TLPreffil
                TLprogdir       ;CD to progdir
                TLProgdir
                (Note: tandem.library also has a routine for putting up an ASL  ↩
                   file
```

```
requester. In order to use that, you must first call TLwindow #-1 [unless
you have already called TLwindow], like this:

  TLwindow #-1
  TLaslfile ......

TLaslfile       ;Put up an ASL file requester
                TLAslfile
                This will be coverred later in the manual).


tandem.librar's General Purpose File Routines

TLopenread, TLopenwrite, TLreadfile, TLwritefile and TLclosefile all use
xxp_hndl to put/get the file handle. Thus if you have 2 files at once, you'd
need to swap the contents of xxp_hndl. You can also use xxp_handl in
dos.library calls, of course; if you close the file other than by means of
TLclosefile, you *must* zeroise xxp_hndl(a4).
```

1.185 oprd

```
                D0 = TLOpenread(A4)

Open a file for reading. Puts handle in xxp_hndl

See also
                File Routines
                Call:  A4 as set by Front.i
       filepath in xxp_buff(a4)

Back:  if good: D0 and xxp_hndl = handle
       if bad:  D0=0, and error code in xxp_errn(a4)

MACRO: TLopenread   no parameters   returns EQ if bad

Example:    [a4 set by Front.i]        or      [a4 set by Front.i]
            move.l xxp_tanb(a4),a6             TLstrbuf #20
            moveq #20,d0                       TLopenread
            jsr _LVOTLStrbuf(a6)               beq Bad
            jsr _LVOTLOpenread(a6)
            tst.l d0
            beq Bad

Sample program:  Tandem/Teaching/24.asm
```

1.186 opwr

```
                D0 = TLOpenwrite(A4)

Open a file for writing. Puts handle in xxp_hndl.

See also
```

```
                File Routines
                Call:  A4 as set by Front.i
       filepath in xxp_buff(a4)

Back:  If good: D0 and xxp_hndl = handle
       If bad:  D0=0, and error code in xxp_errn(a4)

MACRO: TLopenwrite   no parameters   returns EQ if bad

Example:    (same as
                TLOpenread
                ).

Sample program:  Tandem/Teaching/24.asm
```

1.187 wtfl

```
                D0 = TLWritefile(D2,D3,A4)      [
                TLOpenwrite
                 must have been called]

Writes D3 bytes at (D2) to xxp_hndl, which was opened by TLOpenwrite.

See also
                File Routines
                Call:  A4 as set by Front.i
       D2=address to write from   }uses the handle created
       D3=bytes to write          }by TLOpenwrite in xxp_hndl(A4)

Back:  If good: D0=bytes written
       If bad:  D0=0, and error code in xxp_errn(a4)
       n.b. calls TLClosefile if bad

MACRO: TLwritefile \1=addr to write from \2=bytes to write  sets EQ if bad

Example:    [A4 as set by Front.i]     or       [A4 as set by Front.i]
            move.l xxp_tanb(a4),a6              TLwritefile #loc,#siz
            move.l #loc,d2                      beq Bad
            move.l #siz,d3
            jsr _LVOTLWritefile(a6)
            tst.l d0
            beq Bad

Sample program:  Tandem/Teaching/24.asm
```

1.188 rdfl

```
                D0 = TLReadfile(D2,D3,A4)        [
                TLOpenread
                 must have been called]

Read max D3 bytes to (D2) from xxp_hndl; sets D0=bytes read
```

```
See also
                File Routines
                Call:  A4 as set by Front.i
       D2=address to read to       }uses the handle created
       D3=maximum bytes to read    }by TLOpenread in xxp_hndl(A4)

Back:  D0=bytes read (if eof, D0=0, xxp_errn=0, not considered bad);
       if bad, D0=0,
                TLError
                 already called
       n.b. calls TLClosefile if bad

MACRO: TLreadfile  \1=addr to read to \2=max bytes to read  sets EQ if bad

Example:    [A4 as set by Front.i]    or     [A4 as set by Front.i]
            move.l xxp_tanb(a4),a6           TLreadfile #loc,#siz
            move.l #loc,d2                   beq Bad
            move.l #siz,d3                   move.l d0,bytesread
            jsr _LVOTLReadfile(a6)           beq Endof
            tst.l xxp_errn(a4)
            bne Bad
            move.l d0,bytesread
            beq Endof

Sample program:  Tandem/Teaching/24.asm
```

1.189 clos

```
                () = TLClosefile(A4)

Close the file whose handle is in xxp_hndl  (ok to call if already closed)

See also
                File Routines
                n.b. Front.i always calls TLClosefile after Program returns, so if ↩
                    Program
forgets to close xxp_hndl, it will always get closed. If TLreadfile or
TLwritefile are bad, they call TLClosefile before returning.

Call:  A4 as set by Front.i

Back:  xxp_hndl(A4) closed, set to 0

MACRO: TLclosefile    no parameters

Example:    [A4 as set by Front.i]     or     [A4 as set by Fron.i]
            move.l xxp_tanb(a4),a6            TLclosefile
            jsr _LVOTLClosefile(a6)

Sample program:  Tandem/Teaching/24.asm
```

1.190 asdv

```
D0 = TLAssdev(a4)

Finds if an assign exists (without putting up a requester if it doesn't).

Call:  A4 as set by Front.i
       xxp_buff must contain a proposed assign name without the :

Back:  D0<>0 if exists, when buffer contains the first dir assigned to it

MACRO: TLassdev   no parameters     returns NE if exists

Example:   [A4 as set by Front.i]     or      [A4 as set by Front.i]
           move.l xxp_tanb(a4),a6             move.l #'INC ',(a4)
           move.l #'INC ',(a4)                clr.b 3(a4)
           clr.b 3(a4)                        TLassdev
           jsr _LVOTLAssdev(a6)               beq Noinc
           tst.l d0                           bra Yesinc
           beq Noinc
           bra Yesinc

Sample program:  tandem/teaching/42.asm
```

1.191 pfdr

```
               D0=TLPrefdir(D0,A0,A4)

Checks that a use/save prefs dir exists, creates if necessary

Call:   A4 as set by Front.i
        A0=path  (without ENV:/ENVARC:)
        D0=0use,-1save

Back:   D0=-1 if good, dir exists
        D0=0 if bad, xxp_errn(a4)=-10

MACRO:  TLprefdir    \1 = dirname    \2= use or save     (lower case)

Notes:  1. seeks/creates the dir in ENV:, and if D0=-1, in ENVARC:
        2. where it is a sub-dir, call progressively adding 1 element of
           the path at a time. e.g. to create ENV:Tandem/Colors
              (a) 1st call with a0 pointing to   'Tandem',0
              (b) 2nd call with a0 pointing to   'Tandem/Colors',0
        3. Amiga RKM's recommend you make a dir to save your prefs file in
        4. After calling TLPrefdir, you can call
                TLPreffil
                Example:  See
                TLPreffil
```

1.192 pffl

```
D0=TLPreffil(D0,A0,A4)

Saves a prefs file, as Use/Save

Call:   A4 as set by Front.i
        A0=path (without ENV:/ENVARC:)
        D0=0use,-1save
        D2=save from
        D3=bytes to save

Back:   D0=-1 if good
        D0=0 if bad, xxp_errn(a4)=-11

MACRO:  TLpreffil   \1 = path       \2 = use or save    (lower case)
                    \3 = save from  \4 = bytes to save
                    sets EQ if bad (saves all regs)

Example -  save as "Use", prefsz bytes from prefbuf to Mypref/Pref1

        [a4 as set by Front.i]      or     [a4 as set by Front.i]
        move.l xxp_tanb(a4),a6             move.l #'Mypr',(a4)
        move.l #'Mypr',(a4)                move.w #'ef',4(a4)
        move.w #'ef',4(a4)                 clr.b 6(a4)
        clr.b 6(a4)                        TLprefdir a4,use
        moveq #0,d0                        beq Bad
        move.l a4,a0                       move.l #'/Pre',6(a4)
        jsr _LVOTLPrefdir(a6)              move.w #'f1',10(a4)
        tst.l d0                           clr.b 12(a4)
        beq Bad                            TLpreffil a4,use,#prefbuf,#prefsz
        move.l #'/Pre',6(a4)               beq Bad
        move.w #'f1',10(a4)
        clr.b 12(a4)
        move.l a4,a0
        moveq #0,d0
        move.l #prefbuf,d2
        move.l #prefsz,d3
        jsr _LVOTLPreffil(a6)
        beq Bad
```

1.193 pgdr

```
                () = TLProgdir(A4)

Sets the CD to PROGDIR: (unless xxp_tand(a4)<>0 when it does nothing).
Many programs will set the CD to PROGDIR: so that they can load associated
files in the same directory, wherever the user might have placed the
progdir in the file path. But if the program is debugging under Tandem,
then CD'ing to PROGDIR: will CD to Tandem's PROGDIR:, which is not what is
required. (Tandem has a button called "Debug CD" in which you can specify
where Tandem CD's to during debugging - see
                CD to Progdir
                ).

TLProgdir puts the old CD in xxp_cdir(A4), and Front.i will automatically CD
```

```
back to the original CD after  BSR Program  returns.

Call:  A4 as set by Front.i

Back:  CD is at PROGDIR:  (unless xxp_tandm(a4)<>0)

MACRO: TLprogdir

Example:     [A4 as set by Front.i]     or       [A4 as set by Front.i]
             move.l xxp_tanb(a4),a6               TLprogdir
             jsr _LVOTLProgdir(a6)
```

1.194 tuit

```
                Calling TLWindow

the tandem.library routines which appear in the left-hand column of

                How to use tandem.library
                 are designed for a simple,
CLI-like environment. The routines from now on set up a comprehensive
graphical user interface (known as a "GUI" pronounced "gooey").

Before you can call any of the tandem.library routines from now on, you
must call
                TLWindow
                . You can do this as follows:

 1. By calling TLWindow to open a window. The usual.

 2. By calling TLWindow with -1 in D0 (MACRO  TLwindow #-1). This does
    the setting up, but does not actually open a window.

The window(s) you open will be part of a suite of up to 10 windows, numbered
0 to 9. You need not open these in any particular order. Intuition expects
you to use windows for your GUI, so your program should keep track of which
window is active.

tandem.library considers one or another window to be "popped" - if you call
TLWindow, then that window is thereby popped - you will find its number in
xxp_Active(A4), and its xxp_WSuite entry in xxp_AcWind(A4). If no window is
currently popped, then xxp_Active(A4) will be -1.

The various sample programs such as Tandem/Teaching/43.asm illustrate how
to manage a suite of windows.


The Currently Popped Window

Many of the following docs refer to "the currently popped window". You
must read
                The Currently Popped Window
                 carefully before
proceeding.
```

```
Routines for Using Multiple Windows

TLWindow, TLWsub, TLWpop & TLWpoll are designed to make it simple to have
multiple windows (up to 10) on the screen. Each call to TLWindow creates a
window numbered between 0 and 9. You can optionally close any of them with
TLWsub. TLWpop puts any of the open windows into xxp_Active & xxp_AcWind,
where you can call other TL routines to act on that window. TLWpop brings
its window to the front & activates it, but you need not keep it at the
front or active. TLWpoll looks at all open windows, and gets the first
IDCMP message it finds, and pops the window (if not already popped) where
the IDCMP was found.

You can poke and peek to things in xxp_wsuw, pertaining to the window(s) you
create. e.g. to change the pen numbers for TLText to 2,3 in window 4 you
would put:

   move.l xxp_WSuite(a4),a5       ;a5 points to WSuite
   add.w #4*xxp_siz2,a5           ;a5 points to window 4 of WSuite
   move.w #$0203,xxp_FrontPen(a5) ;set pens of window 4 to 2,3

If the window is popped, you can get the xxp_WSuite entry easily like this:

   move.l xxp_AcWind(a4),a5       ;a5 points to popped window
   move.w #$0203,xxp_FrontPen(a5) ;set pens of popped window to 2,3

The number of the
                currently popped window
                 is in xxp_Active(a4). If no window
is currently popped, xxp_Active(a4) = -1.

All the windows tandem.library opens are smart refresh.


Important Note

tandem.library routines do not respond well to you locking intuitionbase.
Also, never place a breakpoint where intuition base is locked, or you will
hang up Tandem's debugger! (Tandem cannot get its screen back). NEVER call
tandem.library routines with multi-tasking disabled, or intuitionbase or
dosbase locked.
```

1.195 popt

```
                The Currently Popped Window

Many tandem.library routines include in their doc "[
                TLWindow
                 must
have been called]". See
                TLWindow
                 for details.

If everything is set up by calling TLWindow, then xxp_Active(a4) is defined.
If xxp_Active(a4)=-1 then no window is currently popped. But if
```

```
xxp_Active(a4) is 0 to 9, then a window is currently popped. That window's
xxp_wsuw structure address is then found in xxp_AcWind(a4) (else, xxp_AcWind
is undefined).

Details of the xxp_wsuw structure are found in Tandem/Support/tanlib.i

The act of opening a window (by TLWindow) causes it to be popped. Calling

                TLWpoll
                 pops whichever window an IDCMP appears in. Calling

                TLWpop
                 pops a nominated window, even if it doesn't have an
IDCMP waiting.

A window can get unpopped by
                TLKeyboard
                 finding an "inactive
window" IDCMP, or by calling
                TLWsub
                 to close it.

Where a tandem.library routine's doc says the routine acts on the currently
popped window, the caller must be sure that xxp_Active is 0-9, not -1.


Example:        [TLWindow has been called]
                [a4 is as set by Front.i]
                ....
                tst.w xxp_Active(a4)       ;is there a popped window?
                bpl.s Popped               ;yes, go
                TLwpoll                    ;else, wait for a window to pop
               Popped:
                move.l xxp_AcWind(a4),a5   ;get popped window's xxp_wsuw
                move.b #1,xxp_FrontPen(a5) ;set its front pen colour
                ....


If a program has multiple windows, then presumably TLWpoll will be followed
by branches to whichever of the windows gets popped, depending on what each
window is used for.

(More sophisticated programs can set up a task for each window, which will
simply wait for its own window to get an IDCMP, instead of using TLWpoll).
```

1.196 wndo

```
                tandem.library window routines

tandem.library has the following window routines:

TLscreen        ;open own screen
                TLscreen
                TLwindow        ;open a window (& set things up)
                TLWindow
```

```
TLwindow0       ;easy call of TLwindow
TLwindow0
TLwsub          ;close a window
TLWsub
TLwclose        ;close everything down
TLWclose
TLwpop          ;choose a window
TLWpop
TLwfront        ;bring window to front
TLWfront
TLreqfull       ;make window full sized
TLReqfull
TLbusy          ;make pointer busy
TLBusy
TLunbusy        ;make pointer unbusy
TLUnbusy

Window Refreshing
TLwupdate       ;update window dims
TLWupdate
TLwcheck        ;see if window resized
TLWcheck
```

1.197 scrn

```
TLscreen                        [
TLWindow
 MUST NOT have been called]
```

```
Attaches A private screen to tandem.library

Normally, tandem.library opens windows & requesters on the default public
screen. To use a private screen, you can use the TLscreen MACRO. This must
be done *before* you call TLWindow or any requesters.

tandem.library must be attached to a screen with at least 4 colours (i.e.
at least 2 bitplanes).

tandem.library routines do not all work on low resolution screens. For
example, TLReqcolor and TLReqshow won't fit on low res screens.

MACRO:   TLscreen
         A4 as set by Front.i
         \1  depth
         \2  title
         \3  pens   (normally DC.L -1)
         Sets EQ if bad

The screen will be high res, non-interlaced, the size of the display clip.
The same number of colours (bit planes) as the default public screen.
If you want to use another type of screen, then analyse TLscreen and
duplicate it with modifications.

Front.i will close your private screen on closedown, but will not close a
```

```
public screen.
```

1.198 wind

```
                D0 = TLWindow(D0-D2,A4)

Opens a window, intialises things if required.

Call:  A4 as set by Front.i
       D0    = window num (0-9)
       D1,D2 = initial position
       D3,D4 = minimum size
       D5,D6 = maximum (& initial) size
               set d5 &/or d6 -ve (not -1) for InnerWidth/InnerHeight
       D7    = flags
       A0    = pointer to title (null delimited string)

Notes:  1. put D0=-1 to initialise everything, without opening a window.
           (D1-D7 and A0 then are not used)

        2. put D7= 0 for a normal resizable, draggable window
                 D7=-1 for a normal resizable, draggable window with scrollers
                 D7= 1 for a normal Unresizable, undraggable window
           (else put d7 = WFLG_ values as per Amiga's intuition/intuition.i)

        3. the window is opened active and popped.

        4. you can use the MACRO
                 TLwindow0
                  for simple
           opening of a default window.

        5. see the flags and IDCMP at the end of the TLWindow subroutine
           in Tandem/Includes/Tandem.i. The IDCMP's are chosen to meet the
           requirements of the various TL routines. You can add more IDCMP's
           with intuition.library calls, but don't take any away if you are
           going to call tandem.library.

        6. if you call TLWindow with d0=-1, you can then look up the screen
           size, &c in xxp_Screen before opening any windows. (e.g. you can
           open bigger windows if it is interlace, &c.)

Back:  D0=0 if bad (i.e. out of memory)

MACRO: TLwindow   \1    = window num (0-9)
                  \2,\3 = initial position
                  \4,\5 = minimum size
                  \6,\7 = maximum (& initial) size
                  \8    = flags
                  \9    = pointer to title (null delimited string)
       Sets EQ if bad

Example:     [A4 as set by Front.i]
             move.l xxp_tanb(a4),a6
             moveq #4,d0
```

```
            moveq #0,d1
            moveq #0,d2
            moveq #100,d3
            moveq #50,d4
            move.l #640,d5
            move.l #200,d6
            moveq #0,d7
            lea title4,a0
            jsr _LVOTLWindow(a6) ;open window 4
            tst.l d0
            beq Bad

        or  TLwindow #4,#0,#0,#100,#50,#640,#200,#0,#title4
            beq Bad

Sample program:   Tandem/Teaching/27.asm

See also:
               TLWsub

               TLWclose
```

1.199 wnd0

```
               TLwindow0 - easy call of
               TLwindow
               TLwindow0 is a MACRO only. It does not take parameters.

To use TLwindow0, you must create a label st_1 which points to a null
delimited string which will become the window's title. e.g.

  st_1: dc.b 'My window',0
   ....
   TLwindow0

TLwindow0 opens a window of minimum width 640, max width = screen width
minimum height 100, maximum height the screen height.

TLwindow0 returns EQ if bad (out of mem).
```

1.200 wsub

```
               D0 = TLWsub(D0,A4)                  [
               TLWindow
                must have been called]

Closes a window opened by TLWindow

n.b. you need not ever call TLWsub for open windows - TLWclose will close
all windows in the suite still open, if any. The window you call TLWsub for
need not be open, however TLWindow must have been called at least once.
```

```
Call:  A4 as set by Front.i
       D0=0 to 9

Back:  window D0 now closed; D0 can be re-used by TLWindow.
       n.b. if window was popped when closed, no window will be popped (i.e.
             xxp_Active(a4) will be -1).

MACRO: TLWsub  \1=window num.

Example:      [a4 as set by Front.i]       or     [a4 as set by Front.i]
              move.l xxp_tanb(a4),a6              TLwsub #2
              moveq #2,d0
              jsr _LVOTLWsub(a6)

Sample program:   Tandem/Teaching/43.asm
```

1.201 wclo

```
               () = TLWclose(A4)              [
               TLWindow
                need NOT have been called]

Closes everything set up by TLWindow.

n.b. This routine need not be called - Front.i calls it automatically when
Program returns. Also, it is ok to call TLWclose if TLWindow was never
called, or if TLWindow failed.

You can call TLWclose to restart everything - e.g. if you want to restart
your program with new preferences. Note that TLWclose closes all resources
opened by TLWindow - it closes all windows, closes all fonts, closes the
screen if it is private, and calls FreeRemember.

Suppose, for example, before your program does anything you put up a
requester to allow the user to select a screen size. You would then call
TLWclose, attach the screen with TLscreen, and continue.

Call:  A4 as set by Front.i

Back:  -

MACRO: TLwclose   no parameters

Example:     [A4 as set by Front.i]      or      [A4 as set by Front.i]
             move.l xxp_tanb(a4),a6              TLwclose
             jsr _LVOWclose(a6)
```

1.202 wpop

```
               () = TLWpop(A4)                    [
               TLWindow
                must have been called]
```

```
Puts a window into xxp_Active. (it's ok to TLWpop a window already popped,
e.g. to bring it to the front & activate it). It then becomes the

                currently popped window
                .

Call:  A4=as set by Front.i
       D0=0 to 9. Window D0 must already exist.

Back:  xxp_Active & xxp_AcWind now have the window, ready for TL calls.
       The window will be brought to the front & activated.

MACRO: TLwpop   \1=window num.

Example:      [a4 as set by Front.i]      or     [a4 as set by Front.i]
              move.l xxp_tanb(a4),a6             TLwpop #3
              moveq #3,d0
              jsr _LVOWpop(a6)

Sample program:   Tandem/Teaching/43.asm
```

1.203 wfro

```
                ()=TLWfront(A4)                    [TLwindow must have been called ↩
                   ]

    Brings the
                currently popped window
                 to the front.
    It's OK to call this if it's already at the front.
    TLWfront does not activate the window.

Call:   A4 as set by Front.i

Back:   -

MACRO:  TLwfront    no parameters

Example:    [a4 as set by Front.i]     or     [a4 as set by Front.i]
            move.l xxp_tanb(a4),a6            TLwfront
            jsr _LVOWfront(a6)
```

1.204 qful

```
                () = TLReqfull(a4)                 [
                TLWindow
                 must have been called]

Sets the
                currently popped window
                 to its full size at posn 0,0.
```

```
Waits for the change to occur before returning.
Does not lock the window, so theoretically it could change again.
Always calls TLWupdate.

Call: A4 as set by Front.i

Back: -

MACR: TLreqfull     no parameters

Example:    [a4 as set by Front.i]      or     [a4 as set by Front.i]
            move.l xxp_tanb(a4),a6             TLreqfull
            jsr _LVOTLReqfull(a6)
```

1.205 busy

```
               ()=TLBusy(A4)

Makes the pointer busy

Call:  A4 as set by Front.i

Back:  -

MACRO:  TLbusy   no parameters

Notes:  1. can call if pointer already busy.
        2. see also
                TLUnbusy
                .

Example:     [a4 as set by Front.i]     or     [a4 as set by Front.i]
             move.l xxp_tanb(a4),a6            TLbusy
             jsr _LVOTLBusy(a6)

Sample Program:  Tandem/Teaching/46.asm
```

1.206 unbs

```
               ()=TLUnbusy(A4)

Makes the pointer unbusy.

Call:  A4 as set by Front.i

Back:  -

MACRO:  TLunbusy    no parameters

Notes:  1. can call if pointer already unbusy.
        2. see also
```

```
                TLBusy
                .

Example:     [a4 as set by Front.i]     or     [a4 as set by Front.i]
             move.l xxp_tanb(a4),a6             TLbusy
             jsr _LVOTLBusy(a6)

Sample Program:  Tandem/Teaching/46.asm
```

1.207 wref

```
                Window Refreshing

tandem.library has two routines for window refreshing:


                TLwupdate

                TLwcheck
                TLMultiline is fully sensitive to window resizing. To make your  ↩
                   programs
sensitive to window resizing, you can use Amiga's _LVOBeginRefresh, &c.
However here is a way that is easier to get working, and seems to work
just as well:

1. tandem.library windows are smart refresh, which means they only ever need
   refreshing if the user resizes them.

2. The drawing routines under
                Primitve Routines
                 will return
   with an error and refuse to draw if the window has been resized.

3. TLKeyboard returns with D0=$96 if an IDCMP sizewindow occurs.

4. Also, before you draw anything, call TLWcheck to see if your window has
   been resized.

5. If any of events 2-4 happen, call TLWupdate immediately.

6. Then if any of 2-4 above happen, call a subroutine which re-draws your
   window. Within that subroutine, get it to jump back to its start
   iteratively, until it gets through drawing the window without any 2-4
   events happening. (This requires you to have a subroutine for each
   re-sizable window, which re-draws that window entirely when 2-4 window
   resizing events happen).

7. Theoretically you should lock intuition base around the redrawing in
   6 above, but I have never found it necessary to do so. It allows better
   multitasking if you don't lock intuitionbase. I suggest you never lock
   intuitionbase or dosbase or call _LVOForbid unless you absolutely have
   to.

(intuition.library SHOULD have a routine LockWindowFrame, which locks
the window size as soon as the user releases the size and zoom gadgets,
```

```
puts a busy clock on the size & zoom gadgets, and returns when the
window's rastport is ready for rendering. Then, the program can issue an
UnLockWindowSize when ready, to return everything to normal).
```

1.208 wupd

```
                () = TLWupdate(A4)                  [
                TLWindow
                 must have been called]

Updates xxp_wsuw data of the
                currently popped window
                .

No need to call TLWcheck first.
Ok to call if window already up to date.
Ok (but pointless) to call for unresizable windows.
TLreqcls and TLWfull call TLWupdate automatically.

Call:   A4 as set by Front.i

Back:   -

MACRO:  TLwupdate    no parameters

See also:
                TLWcheck

                Window Refreshing
                  <-  Important!!

Example:  see
                TLReqbev
```

1.209 wchk

```
                D0 = TLWcheck(A4)                   [
                TLWindow
                 must have been called]

Checks the
                currently popped window
                 has not changed size.

It it often a good idea to put TLWcheck before TLkeyboard calls, with a
branch to redraw if the window is resized. In general, it is not necessary
to call TLReqcls (clear window) before re-drawing it. Just draw over the
top of everything, as long as it's the same as what was there before, and
it will look less "flickery". Do something like this:

Instead of merely:
```

```
     Wait:
      TLkeyboard

Do this:

     Redraw:
      TLwupdate
      bsr DrawWindow

     Wait:
      TLwfront
      TLwcheck
      bne Redraw
      TLkeyboard


Call:   A4 as set by Front.i

Back:   D0=0  if window unchaned
        D0<>0 if window changed

MACRO:  TLwcheck    no parameters   saves all regs    EQ if window unchanged

See also:
                TLWupdate

                Window Refreshing
                  <-  Important!!

Example:  see
                TLReqbev
```

1.210 inot

```
                tandem.library input/output routines

tandem.library has the following input/output routines:


                Input from a window
                TLkeyboard      ;get from keyboard/IDCMP
                TLKeyboard
                TLmget          ;get IDCMP without waiting
                TLMget
                TLwpoll         ;wait for any window
                TLWpoll
                TLwslof         ;slough all queued messages
                TLWslof
                TLwscroll       ;update/read window scrollers
                TLWscroll
                TLgetarea       ;ask user to select an area
                TLGetarea

                Output Text to a window
                TLtext          ;show text
```

```
                TLText
                TLtsize         ;get size &c of text
                TLTsize
                TLtrim          ;show text (trims if off window)
                TLTrim
                TLstring        ;combines TLstrbuf,TLtrim
                TLstring
                (there are lots of other rendering routines elsewhere in this  ↩
                   manual).
```

1.211 iptw

```
                Input from a Window

tandem.library has the following routines to get input (IDCMP) from a
window:

TLkeyboard      ;get from keyboard/IDCMP
                TLKeyboard
                TLmget          ;get IDCMP without waiting
                TLMget
                TLwpoll         ;wait on all windows
                TLWpoll
                TLwslof         ;slough all queued messages
                TLWslof
                TLwscroll       ;update/read window scrollers
                TLWscroll
                TLgetarea       ;ask user to select an area
                TLGetarea
```

1.212 kybd

```
                D0-D3 = TLKeyboard(A4)              [
                TLWindow
                 must have been called]

Waits for an IDCMP to appear at the
                currently popped window
                .

Ignores IDCMP's waiting in unpopped windows.
You can attach help to the <Help> key - see
                Help
                .

Call:  A4 as set by Front.i

Back:  D0 = ASCII if vanilla key
       D1 = mouse X  } relative to the
       D2 = mouse Y  } active window top left
       D3 = bits:  0=shift  3=Ctrl  4=Alt  6=left Amiga  7 = right Amiga
```

```
       The values of D0-D3 are also placed in xxp_kybd+0,4,8,12(a4)
       The complete message as it was received is placed in xxp_megs(a4)

       Or, these special dummy codes:

       $0D     = Return/Enter
       $1B     = Esc
       $80     = left mouse button
       $81-$8A = F1-F10
       $8B     = backspace
       $8C     = tab (left of Q)
       $8D     = Del
       $8E     = up arrow
       $8F     = down arrow
       $90     = right arrow
       $91     = left arrow
       $92     = unused
       $93     = close window
       $94     = gadget up       (see below)
       $95     = menu pick       (see below)
       $96     = window resized
       $97     = inactive window
       $98     = scroller        (see below)
       $99     = boopsi message  (see below)

Notes: (i) tandem.library routines never cause $94 or $99 (gadgetup/boopsi),
           so these will be from things you have attached to windows.

       (ii) if D0 = $95 = menu pick, then
            D1 = menu           0+, -1 if none
            D2 = menu item      0+, -1 if none
            D3 = menu sub-item  0+, -1 if none

      (iii) if D0 = $98 = scroller, then
            D1 = current horizontal top
            D2 = current vertical top
            D3 = object under pointer:  1=left 2=right 2=up 3=down 0=slider
            See
                TLWscroll
                 for details

       (iv) TLKeyboard ignores IDCMP_ACTIVEWINDOW messages

        (v) If xxp_Help(A4) is non-zero, and the user
            presses <Help>, TLKeyboard will display help
            and wait for another message. See

                TLhelp
                 for details.

       (vi) If TLKeyboard gets an IDCMP_INACTIVEWINDOW
            message, it returns $97. You should then
            take appropriate action - e.g. you might
            want to call TLWpoll to wait for another
            window to be active.

      (vii) When you get your message back from
```

```
            TLKeyboard, you may want the details of
            the message as it was received, rather than
            as TLKeyboard processed it into D0-D3. If so,
            you can find a pointer to the original
            message in xxp_mesg(a4), where tandem.library
            stores the most recent gadtools.library
            message collected by TLKeyboard.

     (viii) TLKeyboard gets only key up for keyboard, not
            key down

       (ix) TLKeyboard gets only left mouse down for
            D0 = $80 = lmb click, and discards lmb up
            messages. So, if you want to follow the mouse
            around, you need to keep reading the mouse
            position (perhaps with _LVOWaitTOF's to avoid
            busy wait), and check the window's message
            port until it becomes non-empty. You can
            find the active window's message port
            like this:

            move.l xxp_AcWind(a4),a5  ;a5 = active window's xxp_wsuw
            move.l xxp_Window(a5),a1  ;a1 = window's window structure
            move.l wd_UserPort(a1),a0 ;a0 = active window's message port

            and get the mouse position like this:

            move.l xxp_acWind(a4),a5  ;a5 = active window's xxp_wsuw
            move.l xxp_Window(a5),a0  ;a0 = window's intuition structure
            move.w wd_MouseX(a0),d0   ;d0 = pointer x rel to window
            move.w wd_MouseY(a0),d1   ;d1 = pointer y rel to window

        (x) If TLkeyboard returns an lmb click (D0=$80) thne you will
            usually want too see where it was clicked. The xpos & ypos in
            D1 & D2 will be relative to the topleft of the window, whereas
            all other use of X and Y co-ordinates by tandem.library routines
            is relative to the printable area of the window. So, you will
            almost always do this:

            ;D0 = $80 = lmb click, D1,D2 = mouse x, mouse y from TLkeyboard
             move.l xxp_AcWind(a4),a5   ;a5 = currently popped window
             sub.w xxp_LeftEdge(a5),d1  ;make D1 relative to xpos 0
             bcs discard                ;discard if left border clicked
             cmp.w xxp_PWidth(a5),d1
             bcc discard                ;disacrd if right border clicked
             sub.w xxp_TopEdge(a5),d2   ;make D2 relative to ypos 0
             bcs discard                ;discard if top border clicked
             cmp.w xxp_PHeight(a5),d2
             bcc discard                ;discard if bot border clicked

MACRO: TLkeyboard   no parameters    result in D0-D3

See also:
               TLWpoll
               .

Example - wait for a left mouse click, or quit if close window:
```

```
          [a4 as set by Front.i]      or      [a4 as set by Front.i]
          Waitclick:                          Waitclick:
           move.l xxp_tanb(a4),a6              TLkeyboard
           jsr _LVOTLKeyboard(a6)              cmp.w #$93,d0 ;=close window
           cmp.b #$93,d0                       beq Quit
           beq Quit                            cmp.w #$80,d0 ;=lmb click
           cmp.w #$80,d0                       bne Waitclick
           bne Waitclick

Sample program:    Tandem/Teaching/30.asm
```

1.213 mget

```
                D0,D1,D2,D3 = TLMget(A4)             [
                TLWindow
                 must have been called]

Like TLKeyboard, but merely polls the
                currently popped window
                . Returns as
per TLKeyboard if a message is waiting, else returns immediately with D0=0.


Call:  A4 as set by Front.i

Back:  D0=0 if no message waiting (when D1-D3 undefined)
       If a message received, D0-D3 as per TLKeyboard.
       The values of D0-D3 are also placed in xxp_kybd+0,4,8,12(a4)
       The complete message as it was received is placed in xxp_megs(a4)

MACRO: TLmget   np params
       returns EQ, D0=0, D1-D3 undefined if no message, else mesg in D0-D3

Example:  [A4 as set by Front.i]    or    [A4 as set by Front.i]
          move.l xxp_tanb(a4),a6          TLmget
          jsr _LVOMget(a6)                beq None
          tst.l  d0
          beq None
```

1.214 wpol

```
                D0-D4 = TLWpoll(A4)                [
                TLWindow
                 must have been called]

Polls all open windows, until an IDCMP message is found. It then returns
with D0-D3 as per
                TLKeyboard
                 and if necessary pops the
window.
```

```
Call:  A4 as set by Front.i

Back:  D0-D4 are the results of the IDCMP, viz:
        D0 = code
        D1 = mousex
        D2 = mousey
        D3 = qualifier
        D4 = IDCMP  (presumably, always IDCMP_ACTVIVEWINDOW)
       (the complete message is also found in xxp_mesg(a4)
       The window with the message becomes the
                currently popped window
                .
       (so xxp_Active(a4) will have its window number).
       If more than 1 window has a message polled, the lowest numbered will
           be found first.

MACRO: TLwpoll   no parameters

Example:      [a4 as set by Front.i]      or     [a4 as set by Front.i]
              move.l xxp_tanb(a4),a6             TLwpoll
              jsr _LVOTLWpoll(a6)

sample program:   Tandem/Teaching/43.asm
```

1.215 wslf

```
               ()=TLWslof(A4)                    [
               TLWindow
                must have been called]

Clears all window message buffers. This is useful to get rid of sundry
clicks, resizes &c the user might do while a procedure is happening. You
can then call TLWpoll with a clean slate, after sinalling in some way that
the porcedure is finished (e.g. with
               TLUnbusy
               ).

Call:   A4 as set by Front.i

Back:   -

MACRO:  TLwslof    no parameters

Example:   [a4 as set by Front.i]      or      [a4 as set by Front.i]
           move.l xxp_tanb(a4),a6              TLwslof
           jsr _LVOTLWslof(a6)
```

1.216 wscr

```
               ()=TLWscroll(D0,A4)               [
               TLWindow
                must have been called]
```

```
Get or set the scrollers of the
                currently popped window
                .

If you call
                TLWindow
                 with D7=-1, you will get a window with
scrollers. in each window's xxp_wsuw structure there is a pointer called
xxp_scrl, pointing to an instance of an xxp_scro structure (details in
Tandem/Support/tanlib.i). (If the window has no scrollers, xxp_scrl is 0).

You will see in that where top,visible and total are recorded for the
window's vertical and horizontal slider. The top+visible must be <= the
total. TLWindow initially opens the scrollers with tp=0, vs=256, & tt=256.

See also
                TLKeyboard
                 for getting scroller messages

I hope you like this TLWscroll; it took some doing to get it working!

Call:     D0 = 0  -> re-renders the scrollers to the values in xxp_scrl
          D0 = -1 -> reads the scroller objects to xxp_hztp & xxp_vttp
          (nothing happens if the window has no scrollers)

Return:   scrollers re-rendered or read as requested. All regs saved.

MACRO:    TLwscroll      \1 = 'get' or 'set'

Example:  [A4 as set by Front.i]    or    [A4 as set by Front.i]
          move.l xxp_tanb(a4),a6           TLwscroll get
          moveq #-1,d0                     move.l xxp_AcWind(a4),a5
          jsr _LVOWscroll(a6)              move.l xxp_scrl(a5),a0
          move.l xxp_AcWind(a4),a5         move.l xxp_hztp(a0),d0
          move.l xxp_scrl(a5),a0
          move.l xxp_hztp(a0),d0

Sample program:  Tandem/Teaching/62.asm, 63.asm
```

1.217 geta

```
                D0 =TLGetarea(D0,D1,D2,D3,A0,A4)   [
                TLWindow
                 must have been called]

Allows the user to select an area on the
                currently popped window
                .
The user clicks the left mouse button down, which selects the top left of
the region, and if the pointer moves 0 or more pixels both left and down
a rectangle appears. If the user releases the left mouse button while the
rectangle is showing, the bottom right is also chosen. If the user releases
the lmb while the rectangle is not showing, or touches the Esc key, or if
the window is resized, TLGetarea returns D0 = 0 = cancel.
```

```
Call:   A4 = as set by Front.i
        D0 = minimum xpos  } these are relative to the inside of the window
        D1 = minimum ypos  } border. The rectangle cannot extend beyond these
        D2 = maximum xpos  } limits. If they are invalid, TLGetarea corrects
        D3 = maximum ypos  } them.
        A0 = a 16 byte block of memory to hold the result (it can be in buff)

Back:   D0 = -1 if ok, D0 = 0 if cancel
        if D0 = -1, A0 holds rectangle  0(A0) = top left  } rel to inside
                                        4(A0) = top right } window border
                                        8(A0) = width
                                       12(a0) = height

MACRO:  \1=xpos  \2=ypos  \3=width  \4=height  \5=16-byte result mem
        Returns EQ if cancelled

Example:     [A4 as set by Front.i]    or    [A4 as set by Front.i]
             move.l xxp_tanb(a4),a6           move.l xxp_AcWind(a4),a5
             move.l xxp_AcWind(a4),a5         TLgetarea #0,#0,xxp_PWidth(a5),
             moveq #0,d0                            xxp_PHeight(a5),a4
             moveq #0,d1                      beq Cancel
             move.w xxp_PWidth(a5),d2
             move.w xxp_PHeight(a5),d3
             move.l a4,a0
             jsr _LVOGetarea(a6)
             tst.l d0
             beq Cancel

Sample program:  Tandem/Teaching/59.asm
```

1.218 prtg

```
                Output Text to a Window

tandem.library has the following routines to output (print) to a window:

TLtext          ;show text
                TLText
                TLtsize         ;get size &c of text
                TLTsize
                TLtrim          ;show text (trims if off window)
                TLTrim
                TLstring        ;combines TLstrbuf,TLtrim
                TLstring
                (See also
                Primitive Routines
                 for other routines that
 render to a window}.

These routines work on the
                currently popped window
                 (see
                Calling TLWindow
                ).
```

```
You should keep track of values for the active window's IText, pens, draw
mode, &c. You can set these like this:

  move.l xxp_AcWind(a4),a5         ;a5 = active window's xxp_wsuw
  move.b #$01,xxp_FrontPen(a5)     ;set front pen = 1
  move.b #$00,xxp_BackPen(a5)      ;set back pen = 0
  move.b #RP_JAM2,xxp_DrawMode(a5) ;set draw mode = jam 2
  move.l a4,xxp_IText(a5)          ;point IText to buff
  clr.w xxp_Tspc(a5)               ;set text spacing = 0

The above are the default values TLWindow sets when it opens a window.

For attaching a font and fontstyle to a window, see
                TLNewfont
                .

If you want to change the fontstyle but not the font, then you can poke the
style into xxp_Fsty(a5), but you must also set xxp_Attc(a5) to -1. See

                TLNewfont
                 for font style values.

e.g. to set the font style to bold (= $01 as you'll see in TLNewfont):

   move.l xxp_AcWind(a4),a5   ;a5 = active window's xxp_wsuw
   move.w #1,xxp_FSty(a5)     ;set style to bold
   move.w #-1,xxp_Attc(a5)    ;note style changed

All the tandem.library routines preserve the values of the above things in
the window's xxp_wsuw.
```

1.219 text

```
                () = TLText(D0,D1,A4)              [
                TLWindow
                 must have been called]

Prints text to the
                currently popped window
                .

n.b. D0,D1 are relative to the printing area of the window; xxp_PWidth and
xxp_PHeight show the operative printing area width and height. TLText does
*not* trim the text, to save time. If you want text trimmed, you can use

                TLTrim
                , which automatically checks that everything fits.

See
                Printing to a Window
                  <- Important!

Call:   A4 as set by Front.i
        D0=print xpos
        D1=print ypos
```

```
Back:   -

MACRO: TLtext       \1=xpos  \2=ypos

See also:
                TLstring

                TLTrim
                Example - print string 99 with pens 1,3 at position 20,10:

           [a4 as set by Front.i]     or      [a4 as set by Front.i]
           move.l xxp_tanb(a4),a6             move.l xxp_AcWind(a4),a5
           move.l xxp_AcWind(a4),a5           move.w #$0103,xxp_FrontPen(a5)
           move.w #$0103,xxp_FrontPen(a5)     TLstra0 #99
           moveq #99,d0                       move.l a0,xxp_IText(a5)
           jsr _LVOTLStra0(a6)                TLtext #20,#10
           move.l a0,xxp_IText(a5)
           moveq #20,d0
           moveq #10,d1
           jsr _LVOTLText(a6)

Sample Program:  Tandem/Teaching/28.asm
                 (uses
                TLTrim
                , which works the same as TLText)
```

1.220 trim

```
                TLTrim                              [
                TLWindow
                 must have been called]

Prints text to the
                currently popped window
                .

TLTrim works the same as
                TLText
                , except that it trims the
text to fit the window. If the text is below the bottom of the window, or
only partially fits vertically, it will not be printed. And only those
characters that fit horizontally will be printed.

TLTrim calls
                TLWcheck
                 before printing, and if the window is
resized, it refuses to print. (You would then call TLWupdate & retry).

See
                Printing to a Window
                  <- Important!

Call:   A4 as set by Front.i
        D0=print xpos
```

```
        D1=print ypos

Back:   Like TLText, TLTrim saves all registers.
        It reports the results of trimming as follows:

        If window resized       unprinted                    xxp_errn(a4)<>0
        If didn't fit at all    unprinted                    xxp_errn(a4)=0
        If partially fitted     printed as far as would fit  xxp_errn(a4)=0
        If all would fit        printed fully                xxp_errn(a4)=0

MACRO:  TLtrim   \1 = xpos   \2 = ypos    returns EQ if error

Notes:  1. TLText is a little quicker, so if you are sure the text will fit,
           use TLText.

        2. Other routines that trim, and use xxp_errn in the same way, are:

              TLReqbev     (hence also TLButprt,TLButstr)
              TLReqarea
              TLSlider     (hence also TLSlide)

           in fact, bascically all tandem.library rendering works the same
           way.

        3. Rendering to requester windows never needs to use TLTrim, since
           requester windows are not resizable.
```

1.221 strg

```
               TLstring   MACRO only              [
               TLWindow
                must have been called]

TLstring combines
               TLstrbuf
                and
               TLtrim
               Params:  \1=strnum  \2=xpos  \3=ypos

Example:  TLstrbuf #99       can be written    TLstring #99,#20,#10
          TLtrim #20,#10
```

1.222 tsiz

```
               D4-D7 = TLTsize(A4)                [
               TLWindow
                must have been called]

Calculates the size of text in the
               currently popped window
                pointed to by
xxp_IText.
```

The question of the width of a font is a complex one. The graphics.library routine for font width calulation does not take font spacing into account, so TLTsize corrects that. Also, when fonts are italic, they tip over the edge at the beginning and end. What TLTsize does is this:

1. the width it returns is the total width, including underflow at the start, & overflow at the end.
2. the underflow at start from the most recent call to TLTsize is placed in the
   currently popped window
   's xxp_xmin
3. the overflow at end from the most recent call to TLTsize is placed in the
   currently popped window
   's xxp_xmax

You should always add the xmin (which is mostly 0) to the xpos you want to call TLText/TLTrim at, so that the actual pixels of printing start where you want them. If you are using Jam1 you can advance the print xpos by the total width less xmin less xmax for the next string, but this causes ugly chopping if you use Jam2 (which you usually do). All this is a bit of a pain. Bold font and shadow font may cause xmax to be non-zero, and italic font may cause both xmin and xmax to be non-zero.

if you use
TLreqedit
, then all the above mess is taken care of for you automatically.

Call: A4 as set by Front.i

Back: D4=width also in xxp_wdth(a4)
D5=no. of characters also in xxp_chrs(a4)
D6=font height also in xxp_ysiz(a4)
D7=font baseline also in xxp_basl(a4)

MACRO: TLtsize no parameters

Example - print text in the centre of the window:

```
[a4 as set by Front.i]      or      [a4 as set by Front.i]
move.l xxp_tanb(a4),a6              TLtsize
jsr _LVOTLTsize(a6)                 move.l xxp_AcWind(a4),a5
move.l xxp_AcWind(a4),a5            move.w xxp_PWidth(a5),d0
move.w xxp_PWidth(a5),d0            sub.w d4,d0
sub.w d4,d0                         lsr.w #1,d0
lsr.w #1,d0                         move.w xxp_PHeight(a5),d1
move.w xxp_PHeight(a5),d1           sub.w d6,d1
sub.w d6,d1                         lsr.w #1,d1
lsr.w #1,d1                         TLTrim d0,d1
jsr _LVOTLTrim(a6)
```

Sample program: Tandem/Teaching/33.asm

1.223 murt

```
                tandem.library menu routines

tandem.library has the following menu routines:

TLreqmenu       ;create menu strip (uses TLnm)
                TLReqmenu
                TLnm            ;macro for menu structure
                TLnm
                TLreqmuset      ;attach menu to window
                TLReqmuset
                TLreqmuclr      ;detach menu from window
                TLReqmuclr
                TLonmenu        ;menu item on
                TLOnmenu
                TLoffmenu       ;menu item off
                TLOffmenu
                (note: tandem.library also has
                TLDropdown
                 which does
 drop down menus).

To create a menu, first use a set of TLnm's, to create the basic structure.
Then, call TLReqmenu to convert the TLnm's into a gadtools.library menu
structure. You can then use TLReqmuset and TLReqmuclr to attach and detach
it from the
                currently popped window
                . TLonmenu and TLoffmenu are used to
switch individual menu items on and off.

When the
                currently popped window
                 has TLReqmuset operative,
                TLKeyboard
                will return menu selections as they are made. Refer to TLKeyboard  ↩
                   for
details.
```

1.224 qmen

```
                D0 = TLReqmenu(A0,A4)              [
                TLWindow
                 must have been called]

Sets up a menu, & attaches it to the
                currently popped window
                .
After that, you must call
                TLReqmuset
                 to turn it on.

See also
                Menu Routines
```

```
                TLnm
                Call:  A4 as set by Front.i
        A0=a NewMenu structure. All nm_Title values less than 1024 are
                assumed to be string numbers. These are replaced by
                pointers to the strings, ready for gadtools CreateMenus.
             If any non-zero gnm_CommKey's are <1024, use the same string
                no. for each, and within that string, the CommKeys in
                order. ReqMenu then fills them in in order.
             You can use the
                TLnm
                 MACRO to create the newmenu

Back:  D0=0 if bad (i.e. gadtool _LVOCreateMenus fails)
       xxp_Menu is now ready for TLReqmuset

MACRO: TLreqmenu    \1=pointer to NewMenu    sets EQ if bad

Example:  See Tandem/Teaching/38.asm
```

1.225 qnmn

```
TLnm      The TLnm MACRO makes it easy to make a NewMenu structure.

MACRO       TLnm references take 3,4 or 5 parameters, as follows:
            \1 is 1,2,3 or 4 for NM_TITLE,NM_ITEM,NM_SUB, or NM_END
            \2 is a string number, or -1 for NM_BARLABEL (you can have
                    a pointer instead of a string number if you wish)
            \3 if present, is the string number of the CommKeys string
                    (you can also have a pointer if you wish)
            \4 if present, is the flags           } usually omitteed
            \5 if present, is the mutual exclude  } (when 0 assumed)
            Teaching/37 has an example of a NewMenu list using TLnm
            The TLnm MACRO is appended to the IncAll.i file

For the gnm_CommKeys, put all the commkeys in order in a string, and put its
stringnum for \3 wherever there is a commkey; TLReqmenu replaces \3 by
a pointer to that string, which it bumps for each commkey encountered.

For all the \2's, simply put string numbers. TLReqmenu relaces them by
pointers.

You can if you wish use pointers direct instead of string numbers for \2
and \3. Pointers cannot be <1024, so TLReqmenu will only assume that they
are string numbers if <1024, and if not, will leave them alone.

This system makes your program re-runnable, even though TLReqmenu changes
DC constants, because when TLReqmenu first changes the string numbers to
pointers, the pointers remain valid if your program exits but remains in
memory - this therefore doesn't stop you from setting the PURE bit for your
program. (Normally, programs that change their own data areas cannot have
their PURE bit set); but your program would not be ROM'able.

Example:    See teaching/38.asm
```

1.226 qmus

```
                () = TLReqmuset(A4)               [
                TLReqmenu
                 must have been called]

Switches on the menu attached to the
                currently popped window
                 by TLReqmenu.
It remains on until
                TLReqmuclr
                 is called.

While the menu is attached, menu selections are picked up by
                TLKeyboard
                .
The TLReqmuset & TLReqmuclr pair can be called repeatedly.
It is ok to call TLReqmuset if the menu is already attached.

See also
                Menu Routines
                Call:  A4 as set by Front.i

Back:  -

MACRO: TLreqmuset     no parameters

Example:    See teaching/38.asm
```

1.227 qmuc

```
                () = TLReqmuclr(A4)               [
                TLReqmenu
                 must have been called]

Switches off the menu attached by TLreqmuset.

TLWclose/TLWsub call TLReqmuclr automatically if needed, when closing the
window.

It is ok to call TLReqmuclr if the menu is already switched off.

Call:      A4 as set by Front.i

Back:      -

MACR: TLreqmuclr     no parameters

Example:    See teaching/37.asm
```

1.228 onmn

```
                ()=TLOnmenu(D0,D1,D2,A4)              [
                TLReqmenu
                 must have been set]

Re-enables a menu item on the
                currently popped window
                .

See also
                Menu Routines
                It is ok to call this whether TLReqmuset or TLReqmuclr is  ↩
                   operatvie.
It is ok to call TLOnmenu for an item that is alreay switched on.

Call:   A4 as set by Front.i
        D0=menu number
        D1=item number
        D2=sub-item number

Back:   -

MACRO:  TLonmenu  \1=menu  \2=item  \3=sub-item
```

1.229 ofmn

```
                ()=TLOffmenu(D0,D1,D2,A4)         [
                TLReqmenu
                 must have been called]

Disables a menu item on the
                currently popped window
                .
It is ok to call this whether TLReqmuset or TLReqmuclr is operatvie.
It is ok to call TLOffmenu for an item that is alreay switched off.

See also
                Menu Routines
                Call:   A4 as set by Front.i
        D0=menu number
        D1=item number
        D2=sub-item number

Back:   -

MACRO   TLoffmenu  \1=menu  \2=item  \3=sub-item
```

1.230 ftrt

```
                tandem.library rendering routines

tandem.library has the following rendering routines:
```

```
                Font routines
                TLgetfont       ;specify font
                TLGetfont
                TLnewfont       ;attach font
                TLNewfont
                TLfsub          ;close font
                TLFsub

                Miscellaneous Rendering
                TLreqbev        ;make a bevelled box
                TLReqbev
                TLellipse       ;draw an ellipse
                TLEllipse
                TLreqarea       ;make a coloured area
                TLReqarea
                TLreqcls        ;clear a window
                TLReqcls
                TLgetilbm       ;load an ilbm from a diskfile
                TLgetilbm
                TLfreebmap      ;free a bitmap
                TLfreebmap
                TLputilbm       ;save an ilbm to a diskfile
                TLputilbm
                TLresize        ;resize an area of a rastport
                TLresize
                (for text rendering see
                Printing Text to a Window
                ).
```

1.231 fntr

```
                Font Routines

tandem.library has the following font routines which are discussed here:

TLgetfont       ;specify font
                TLGetfont
                TLnewfont       ;attach font
                TLNewfont
                TLfsub          ;close font
                TLFsub
                The following are discussed later in the manual:

TLaslfont       ;select a font from FONTS:
                TLAslfont
                TLreqfont       ;manage the font suite
                TLReqfont
                How tandem.library Deals with Fonts

(Where you see xxp_Fnum in the xxp_wsuw structure, you should not poke a
 font number - to attach a font to a window, use TLNewfont. So, in other
 words, xxp_Fnum is effectively read only. You can poke a value to xxp_Fsty,
 but you must at teh same time poke -1 to xxp_Attach. For more on this, see
```

```
                Printing Text to a Window
                , and read on below).
```

tandem.library starts off with the default font attached to each window it creates.

tandem.library's system for fonts works like this: tandem.library when things are set up creates an instance of an xxp_fsui structure (see in Tandem/Support/tanlib.i) which is pointed to by xxp_FSuite(a4). This allows for 10 fonts, numbered 0-9, and each font can be half height, double width, or both, theoretically up to 40 fonts. Font 0 is always Topaz/8, and is attached by default to every window's xxp_wsuw.

xxp_fsui also has fonts 10-11, which are reserved as default TLReqshow & requester fonts. See TLPrefs.

Whichever font is attached to a window's wsuw is the font used by TLText, TLTrim, and TLReqedit. A window also has attached to it two other fonts, which are also by default Topaz/8. These are passed on to any requesters or context sensitive help opened by that window.

Each of the three fonts I mentioned above (the window font, the requester font and the help font) is one of fonts 0 to 11 in the xxp_FSuite fonts. The three fonts may be set in the window's wsuw to any combination of these styles:

```
 plain   bold   italic   super/subscript   double width   shadow font
```

along with various styles of underlining and overlining. When TLText etc are required to render one of fonts 0-9 in super/subscript or double width or both, tandem.library develops the required font and places a pointer to it in the xxp_FSuite fonts. Thus, there can (theoretically) be up to 40 fonts in xxp_FSuite, numbered 0-9, each in plain, half ht &/or double width. tandem.library does bold, italic, shadow, and underlining not by opening new fonts but by the way it calls the graphics.library font rendering.

You will also see in the xxp_wsuw that text spacing may be specified for the window, requester and help fonts. Text spacing is normally zero, but non-zero values may be useful for large fonts.

As a quick summary:

```
TLGetfont puts a predetermined font & fontsize in font 1-9

TLAslfont lets the user to select a font & fontsize to put into
                     font 1-9

TLnewfont attaches a font number & fontstyle to window's
                     window font, requester font, or help font.

TLfsub    closes one of font 1-9 (0 is always open, 10-11 may be
                     changed by
                TLPrefs
                 but not closed)

TLReqfont allows the user to open/close & change fonts 0-9, i.e. to
```

```
                    manage the suite of fonts.
```

1.232 gtft

```
                D0 = TLGetfont(D0,D1,A0,A4)        [
                TLWindow
                 must have been called]

Puts a predetermined font & and fontsize into font 1-9

Notes:  1. If there was already a font there, TLFsub will be called for it
        2. The font num is 1-9; font 0 is reserved for Topaz/8
        3. The font does not get opened - it will be opened for whichever
           style, when TLNewfont first uses it.
        4. If the font was already there, and it had been attached to any
           windows, then further rendering to those windows will use the
           font now attached.

See also
                Font Routines

                TLAslfont
                Call:  A4 as set by Front.i
       A0=fontname               e.g. 'Times.font',0
       D0=font number (1-9)
       D1=font height            e.g. 24

Back:  -      (does not open the font. TLNewfont does that if required)

MACRO: TLgetfont    \1=fontname  \2=number  \3=height

Example:   see
                TLNewfont
```

1.233 newf

```
                D0 = TLNewfont(a4)                 [
                TLWindow
                 must have been called]

Sets the font and style for the
                currently popped window
                , or its requesters
or help.

See also
                Font Routines
                Font style

You can use the the following style bits, to specify the style of font
rendering (these are documented in tandem/support/tanlib.i):
```

```
   (a) bit 0    bold
   (b) bit 1    italic
   (c) bit 2-5  0001 underline          0110 double underline
                0010 superscript        0111 double underline + overline
                0011 overline           1000 dotted underline
                0100 subscript          1001 strike through
                0101 under + overline
       (some of these need reasonably-sized text to display meaningfully)
   (d) bit 6    shadow font
   (e) bit 7    double width font

You can use any combination of the above, e.g.:

 move.w #xxp_xbit0!xxp_xbit7,d1 ;style = double width + bold

The double underlining, and overlining, look better with larger fonts.

To change font style for a window (or its requesters or help), you can
poke a new value into xxp_Fsty, xxp_RFsty, or xxp_HFsty. But you must
also poke -1 to xxp_Attach.

You may however poke new text spacing values into Tspc, RTspc or HTspc, e.g.

 move.l xxp_AcWind(a4),a5   ;set text spacing to 3 (normally 0)
 move.w #3,xxp_Tspc(a5)

You can also poke new values into FrontPen or BackPen, e.g.

 move.l xxp_AcWind(a4),a5   ;set backgound pen to 11
 move.b #11,xxp_BackPen(a5)

Shadow font uses xxp_shad in the window's xxp_wsuw entry. You will see the
pen number for the shad (default 2) in xxp_shad+1, the y displacement
(default 1) in xxp_shad+2, and the x displacement (default 2) in xxp_shad+3.

You can poke new values for these, e.g.

 move.l xxp_AcWind(a4),a5
 move.b #7,xxp_shad+1(a5) ;use pen 7 for shadow font
 move.b #3,xxp_shad+3(a5) ;make x displacement 3


Call:  A4 as set by Front.i
       D0 = font number (0-11)
       D1 = style (see above)
       D2 = where (0=window, 1=requesters, 2=help)

Back:  D0<>0 if good
       D0=0 if bad, when  xxp_errn(a4)=3 diskfont.library unopenable
                          xxp_errn(a4)=4 can't find/open that font

Notes: 1. If D0 is 1 to 9, TLGetfont or TLAslfont must have been called for
          that font.
       2. The first time any font/style is called, the font is opened with
          that style. If that fails, Topaz/8 will be attached. To check
          that didn't happen, read Fnum for that window.
       3. The first time a font number is called, in either normal size,
```

```
          super/sub script, or double width, or both sup/sub + double width,
          tandem.library must open the font. This can fail, see point 2.
          But subsequent calls of TLNewfont cannot fail.
       4. If you set D1=1, that font will be used for requesters opened
          while that window is the
                currently popped window
                , over-riding
          the
                TLPrefs
                 font & fontstyle. If you set D1=2,
          then context-sensitive help opened under that window will be
          affected. (Note: if a requester or help window tries to open,
          but won't fit, tandem.library will also try to open it with
          font 0).

MACRO: TLnewfont    \1=as D0 above  \2=as D1 above  [\3=as D2, default 0]
       sets EQ if bad

Example - attach Times/24, bold to font 3 and
                currently popped window
                       string 20 is 'times.font',0
       [a4 as set by Front.i]       or       [a4 as set by Front.i]
       move.l xxp_tanb(a4),a6                TLstra0 #20
       moveq #20,d0                          TLgetfont a0,#3,#24
       jsr _LVOStra0(a6)                     TLnewfont #3,#1,#0
       moveq #3,d0                           beq Bad
       moveq #24,d1
       jsr _LVOGetfont(a6)
       moveq #3,d0
       moveq #1,d1
       moveq #0,d2
       jsr _LVONewfont(a6)
       tst.l d0
       beq Bad

Sample programs:   Tandem/Teaching/32.asm, Tandem/Teaching/35.asm
```

1.234 fsub

```
                ()=TLFsub(D0,A4)                    [
                TLWindow
                 must have been called]

Closes a font.
It's ok to call this if that font number is already closed / never opened.

Notes: 1. TLFsub need never be called. Front.i closes all open fonts
          at shutdown time. If you have temporarily used a very large font,
          close it to save memory.
       2. All windows which have the font that closes, if any, will have the
          font replaced by font 0 (i.e. Topaz/8).

Call:  A4 as set by Front.i
       D0=font num to be closed (1-9). (font 0 must not be closed)
```

```
Back:  -

MACRO: TLfsub      \1 = fontnum (1-9)
```

1.235 rndr

```
                Other Rendering Routines

tandem.library has the following miscellaneous rendering routines:

TLreqbev        ;make a plain or bevelled box
                TLReqbev
                TLellipse       ;draw an ellipse
                TLEllipse
                TLreqarea       ;make a coloured area
                TLReqarea
                TLreqcls        ;clear a window
                TLReqcls
                TLgetilbm       ;load an ilbm from a diskfile
                TLgetilbm
                TLfreebmap      ;free a bitmap
                TLfreebmap
                TLputilbm       ;save an ilbm to a diskfile
                TLputilbm
                TLresize        ;resize an area of a rastport
                TLresize
                All the above rendering routines will trim their output to fit if  ↩
                   the window
is too small, and will refuse to execute if the window is resized. See

                TLtrim
                 for details.

Most of these routines can write to a rastport, rather than the currently
popped window, if desired. You can thus set up a bitmap & rastport, render
to it, and then copy portions of it to the window as required.
```

1.236 qbev

```
                () = TLReqbev(D0-D3[,D4-D5],A4)  [
                TLWindow
                 must have been called]

Draws a bevelled or plain box in the
                currently popped window
                .
The box has top & bottom 1 pixel high, and sides 2 pixels wide.

See also
                Primitive Rountines
                Call:  A4 as set by Front.i
       D0=xpos                  set bit 31 of D0 for recessed
```

```
                                    bit 30 of D0 for unbevelled box
                                    bit 29 of D0 for d4,d5 = dark,light pens
       D1=ypos                   set bit 31 of D1 for rastport in A0
       D2=width
       D3=height
       D4,D5 = dark,light pens if bit 29 of D0 set

Back:  all regs saved
       xxp_errn(a4)<>0 if bad (i.e. window resized)
       if the box won't fit, TLREqbev draws as much as will fit

MACRO: TLreqbev      \1=xpos  \2=ypos  \3=width  \4=height
                     \5=rec   if recessed
                     \5=box   if unbevelled
                     \6=rastport  (else
                 currently popped window
                 )
                     \7=dark pen if bevelled, or pen if unbev  (default 1)
                     \8=light pen if bevelled                  (default 2)
                     (any of 5-8 can be omitted or null)
       sets EQ if bad, i.e. window resized


Example:     [a4 as set by Front.i]      or      [a4 as set by Front.i]
             move.l xxp_tanb(a4),a6              Try:
             moveq #20,d0                         TLreqbev #20,#10,#40,#20
             moveq #10,d1                         bne Drawn
             moveq #40,d2                         cmp.l #11,xxp_errn(a4)
             moveq #20,d3                         beq Bad
            Try:                                  TLwupdate
             jsr _LVOTLReqbev(a6)                 bra Try
             cmp.l #11,xxp_errn(a4)
             bcs Drawn
             beq Bad
             jsr _LVOTLWupdate(a6)
             bra Try

        See also Teaching/29.asm for combinations of bevs &c
        (bevelled box drawing - a new art form?)
```

1.237 elps

```
               (D0)=TLEllipse(D0-D7/A0-A1/A4)     [
               TLWindow
                must have been called]

Draws an ellipse on the
               currently popped window
               , with clipping if required.
drawing will not proceed, or will not be finished, if the window is resized.

If you set bit 31 of D1, then A0 will be a rastport, else draw to the

               currently popped window
               . xxp_FrontPen, or the APen of the rastport, will be
```

```
used.

Call:  A4  as set by Front.i
       D0  x centre           bit 31 of D0 set if solid, else outline only
       D1  y centre           bit 31 of D1 set if A0 = rastport
       D2  x radius
       D3  y radius
       D4  xmin       }
       D5  ymin       }  limits of drawing area
       D6  xmax       }
       D7  ymax       }
       A0  rastport if D1 bit 31 set, else ignored

Back:  All regs saved.
       Sets xxp_errn(A4)<>0 if bad (window resized)

MACRO: TLellipse \1 - \8  same as D0-D7
                 \9 = rastport
                 \10 = 'solid'
                 \9,\10 may be omitted or null
                 returns EQ if bad

Sample prgram:   Tandem/Teaching/58.asm
```

1.238 qare

```
                () = TLReqarea(D0-D4,A4)           [
                TLWindow
                 must have been called]

Draws a coloured area in the
                currently popped window
                , or if bit 31 of D1 is
set, to a rastport.

If bit 29 of D0 is set, the pen will be in D4, else uses xxp_FrontPen of

                currently popped window
                , or the APen of the rastport.

If width & height are too large to fit, as much as will fit will be drawn.

Call:  A4 as set by Front.i
       D0  xpos            bit 29 of D0 is set if the pen is in D4
       D1  ypos            bit 31 of D1 is set if A0 is the rastport
       D2  width
       D3  height
       D4  pen if bit 29 of D0 set, else ignored
       A0  rastport if bit 31 of D1 set, else ignored

Back:  All regs saved.
       Sets xxp_errn(A4)<>0 if bad (window resized)

MACRO: TLreqbev      \1=xpos  \2=ypos  \3=width  \4=height
                     [\5=pen]
```

```
                    returns EQ if bad


Example:     [a4 as set by Front.i]      or      [a4 as set by Front.i]
             move.l xxp_tanb(a4),a6              Try:
             moveq #20,d0                         TLreqbev #20,#10,#40,#20,#3
             moveq #10,d1                         bne Drawn
             moveq #40,d2                         TLwupdate
             moveq #20,d3                         bra Try
             moveq #3,d4                         Drawn:
            Try:
             jsr _LVOTLReqbev(a6)
             tst.l xxp_errn(a4)
             beq Drawn
             jsr _LVOTLWupdate(a6)
             bra Try
            Drawn:
```

1.239 qcls

```
                () = TLReqcls(A4)                  [
                TLWindow
                 must have been called]

Clears the
                currently popped window
                .

Always calls TLWupdate before clearing.

Call:   A4 as set by Front.i

Back:   -

MACRO:  TLreqcls      no parameters

Example:    [a4 as set by Front.i]    or     [a4 as set by Front.i]
            move.l xxp_tanb(a4),a6           TLreqcls
            jsr _LVOTLReqcls(a6)
```

1.240 gilb

```
                D0=TLGetilbm(D0-D6,A0-A1,A4)       [
                TLWindow
                 must have been called]

Loads an ILBM file into a bitmap.

The bitmap is created by TLGetilbm, and it may be released after it is
finished with by passing its address to TLfreebitmap.

IF you set bit 31 of D1 (= load into public mem), then you must NOT use
```

```
the "bitmap" in a rastport, else you can create a rastport & attach to bitmap
to it.

Call:  A4 as set by Front.i
       file path in buff
       A1 = address of a 790 byte buffer to hold BMHD and CMAP
       {D0 = -1 if load BMHD, CMAP and BODY
       {D0 = 0  if load BMHD, CMAP only
       D1 = max number of planes to load.
            set bit 31 of D1 to load into public mem, else loads into chip

       9. loaded into A1:
            bytes 0-19   the BMHD                 } Total 790 bytes if
            bytes 20-21  the CMAP size in bytes   } 8 bitplanes
            bytes 22+    the CMAP                 }

Back:  A0 = address of bitmap with BODY, if successful
       Loaded into A1 will be:  bytes 0-19   the BMHD
                                bytes 20-21  the size of the CMAP
                                bytes 22+    the CMAP, consisting of 3 bytes
                                             for each colour, max 256 colours
       If bad, or D0 was 0 on call, A0 unchanged
       xxp_errn(A4)<>0 if bad

MACRO: TLgetilbm  \1  max planes
                  \2  790 byte buffer address
                  [\3 'nobody'] if BODY not to be loaded
                  [\4 'public'] if public memory to be used
                  \3, \4 may be null or omitted
                  returns EQ if bad

Sample program:  Tandem/Teaching/52.asm
```

1.241 fbmp

```
               TLfreebmap   (MACRO only)

Frees a bitmap you have created by calling _LVOAllocVec for the Bitmap
and all its bitplanes. You must NOT call TLfreebmap if you have used
AllocRemember or AllocMem to reserve memory for the bitmap or its bitplanes
(n.b. TLpublic and TLchip use TLAllocRemember NOT AllovVec).

The main use of TLfreebmap is to release a bitmap you have created with

               TLgetilbm
               , which uses AllocVec.

MACRO:  \1=bitmap address

Example:       [A4 as set by Front.i]
               TLfreebmap #bmap        ;(bmap was returned by TLgetilbm)
```

1.242 pilb

```
                D0=TLPutilbm(D0-D3,A0,A4)          [
                TLwindow
                 must have been called]

Put a region of the
                currently popped window
                 to an ILBM file.
Makes a CMAP from xxp_Screen.
If bit 31 of D0 is set, saves a region from a rastport.

If from a window, aborts if window resized (if file partly complete, deletes
it).

Call:  A4 as set by Front.i
       filepath in buff
       D0,D1  top left              bit 31 of D0 set if from bitmap
       D2,D3  width, height
       A0     rastport, unused if window

Back:  D0 = 0 if bad

MACRO: \1,\2 = topleft  \3,\4 = width, height  \5=bitmap
       returns EQ if bad

Sample program:  Tandem/Teaching/62.asm
```

1.243 resz

```
                D0=TLResize(A4,D0-D4,A0)        [
                TLWindow
                 must have been called]

Resizes a region of the
                currently popped window
                . If bit 31 of D0 is set,
resizes a region of a rastport.

Call:  A4 as set by Front.i
       D0,D1  top left                  bit 31 of D0 set if from bitmap
       D2,D3  width, height
       D4     pen to fill exposed areas
       A0     rastport, unused if window

Back:  D0=0 if bad

MACRO: \1,\2 = top left  \3,\4 = width,height
       \5 = pen to fill exposed areas
       if bit 31 of \1 set, \6 = rastport
       sets EQ if bad

Example program:  Tandem/Teaching/57.asm
```

1.244 reqx

```
                Requester Routines

tandem.library has the following requester routines:

TLAslfile       ;asl file requester
                TLAslfile
                TLAslfont       ;asl font requester
                TLAslfont
                TLReqchoose     ;put up a choose-type requester
                TLReqchoose
                TLReqcolor      ;put up a palette-type requester
                TLReqcolor
                TLReqinfo       ;put up an info-type requester
                TLReqinfo
                TLReqinput      ;put up an input-type requester
                TLReqinput
                TLReqshow       ;put up a show-type requester
                TLReqshow
                TLPrefs         ;set prefs for amiga.library GUI
                TLPrefs
                TLData          ;put up a temporary data window
                TLData
                TLProgress      ;draw a progress bar
                TLprogress
                TLReqfont       ;font suite requester
                TLReqfont
                TLMultiline     ;use window to edit a txt buffer
                TLMultiline
                TLattach        ;attach memory to TLMultiline
                TLattach
                TLhelp          ;context sensitive help
                TLhelp
                tandem.library has a wide variety of requesters. They are all  ↩
                   fully font
sensitive. The requesters generally open their own windows.

Here are the requester types:

     TLReqchoose      ;for selecting from a range of options
     TLReqcolor       ;for choosing a pen or palette
     TLReqinfo        ;for displaying static information
     TLReqinput       ;for inputting a string or number
     TLReqshow        ;for displaying dynamic information
     TLReqfont        ;for organising a suite of fonts
     TLhelp           ;for online context sensitive  help

These are not requesters, but similar to requesters:

     TLData           ;for temporary display of information
     TLProgress       ;for reporting progress on a task

Also covered here is

     TLPrefs          ;for setting tandem.library preferences
```

All these except TLProgress use their own little window, which closes when they return. All except TLData and TLProgress operate synchronously - i.e. when you call them, everything else stops until they return (but you can still use intuition gadgets like depth arranging, &c). (They do not disable multitasking, however - tandem.library never disables multitasking. So of course you can make requesters asynchronous by using separate tasks for other windows).

If a window is
currently popped
, the requesters try to open within that window, positioned according to xxp_reqx and xxp_reqy in the window's xxp_wsuw structure, and using the xxp_RFnum font. If no window is popped, the requesters will open at the top left of the screen, using Topaz/8. You can call any of the requesters before you open any windows, if you want.

You do not call TLHelp directly - see
Help
.

Getting the Requester Size

You can if you wish call any of the TL requesters with xxp_ReqNull(a4) set to 0. This causes them to abort after calculating the requester size, which you may find in xxp_reqw and xxp_reqh. You can then adjust xxp_reqx and xxp_reqy in the light of that information, for aesthetic positioning (see Tandem/Teaching/39.asm for an example). If you call a requester with xxp_ReqNull=0, the Requester will also set it back to -1 before returning. (See Tandem/Teaching/39.asm for an example).

Advanced Usage - Adjusting Requester Appearance

For TLReqchoose,TLReqinfo and TLReqinput, you can poke addresses of subroutines into xxp_Hook0, xxp_Hook1 and xxp_Hook2, to change the appearance of the requester as follows:

1. After tandem.library calculates the dimensions and position of the requester, i.e. where it might return if xxp_ReqNull is zero, you can have xxp_ReqNull non-zero as usual, and instead of the usual zero in xxp_Hook1, you can have the address of a subroutine. That subroutine can, if it wants, change the position of the requester, and also increase (but not decrease) its size. tandem.library will still draw all except the background of the requester window the same size, but since you have enlarged it, you can leave room to draw, say, a logo in the requester. tandem.library does not check the validity of the changes you make to xxp_reqx, xxp_reqy, xxp_reqw and xxp_reqh

2. Then, after tandem.library opens the requester window, but before it puts anything on it, if xxp_Hook1 is non-zero, it will call the address therein. This enables you to say put a background picture on it. Tandem will already have set the xxp_but.. information, so you can also fiddle with these to spread out the buttons, or make them bigger &c.

3. Then, after the requester is drawn, but before it waits for input from

```
   the user, tandem.library calls the address in xxp_Hook2 if non-zero.

4. Finally, all the above hooks are zeroised.

5. When any of the above requesters first open they MOVEM.L D0-D7/A0-A6,-(A7)
   and then  MOVE.L A7,xxp_Stak(A4). Thus, when your hooks are called, you
   can find the register values that you called the requester with. (The
   requesters call the hook with A4 set to the Front.i value, but meaningless
   values in all the other registers).

See Tandem/Teaching/51.asm for an example program.
```

1.245 aslf

```
                D0 = TLAslfile(D0,D1,A0,A1,a4)     [
                TLWindow
                 must have been called]

Puts up an ASL requester to select a filename

see also
                File Routines
                Call:  A4 as set by Front.i
       D0=strnum of hail
       D1= -1save +1load
       A0=filename  (in a 32 byte buffer)   }The initial values are used
       A1=dirname   (in a 130 byte buffer)  }as a prompt in the requester

Back:  D0<>0 if good, when A0=filename, as updated
                           A1=dirname, as updated
                           filepath in xxp_buff (ready for TLOpenread/write)
       D0=0 if bad,  when xxp_errn(A4)=0 if user cancelled, else
                          xxp_errn(A4) has
                error code
                .

MACRO: TLaslfile    \1=filname  \2=dirname  \3=hail string num  \4=sv/ld

Example:   (hail in string 10)       or      (hail in string 10)
           [a4 set by Front.i]                [a4 set by Front.i]
           move.l xxp_tanb(a4),a6             TLaslfile #dir,#fil,#10,sv
           moveq #10,d0                       bne Good
           moveq #-1,d1                       tst.l xxp_errn(a4)
           lea dir,a0                         beq Cancel
           lea fil,a1                         bne Bad1
           jsr _LVOTLAslfile(a6)             Good:
           tst.l d0                           TLopenwrite
           bne.s Good
           tst.l xxp_errn(a4)
           beq Cancel
           bne Bad1
          Good:
           jsr _LVOOpenwrite(a6)
```

```
Sample program:  Tandem/Teaching/37.asm
```

1.246 aslt

```
               D0 = TLAslfont(a4)                [
               TLWindow
                must have been called]

Puts up an ASL requester to select a font (then as per TLGetfont).

See also
               Font Routines

               TLGetfont
               Call:  A4 as set by Front.i
       D0=font num (1-9)

Back:  D0<>0 if good
       D0=0 if bad, when  xxp_errn(a4) = 0 if user cancelled
                          else xxp_errn(a4) has
               error code
               .

MACRO: TLaslfont    \1=font num (1-9)    sets EQ if bad

Example:   [a4 as set by Front.i]    or     [a4 as set by Front.i]
           move.l xxp_tanb(a4),a6             TLaslfont #1
           moveq #1,d0                        bne Good
           jsr _LVOTLAslfont(a6)              tst.l xxp_errn(a4)
           tst.l d0                           beq Cancelled
           bne Good                           bne OutOfMem
           tst.l xxp_errn(a4)
           beq Cancelled
           bne OutOfMem

Sample program:   Tandem/Teaching/24.asm
```

1.247 qcho

```
               D0 = TLReqchoose(D0-D1,A4)        [
               TLWindow
                must have been called]

Puts up a list of options for the user to choose from.
The user can click a box beside each item, or press a function key.

See also
               Requester Routines
               .

Call:  A4 as set by Front.i
       D0=string num of header  } There must be D1+1 consecutive strings,
```

```
       D1=no. of choices (1+)   } starting with string D0 = header
       But, if D1=0, D0 is ignored, & buff contents put up with "Click
                  to acknowledge".

Back:  D0=choice, 1+, if good
       D0=0 if bad, when xxp_errn(a4) has
                error code
                .

MACRO: TLreqchoose  [ \1=strnum of header,etc  \2=choices]  sets EQ if bad.
       But if parameters omitted, as per Call: above with D1=0

Examples: 1. header, 5 choices in strings 20-25:

                [a4 as set by Front.i]      or     [as set by Front.i]
                move.l xxp_tanb(a4),a6             TLReqchoose #20,#5
                moveq #20,d0                       beq Bad
                moveq #5,d1
                jsr _LVOTLReqchoose(a6)
                tst.l d0
                beq Bad

             2. report what TLError turns up   or     TLerror
                move.l xxp_tanb(a4),a6             TLreqchoose
                jsr _LVOTLError(a6)
                moveq #0,d1
                jsr _LVORLReqchoose(a6)

Sample program:   Tandem/Teaching/35.asm
```

1.248 qcol

```
                D0 = TLReqcolor(A4)            [Needs OS3.0+ (library version 39+) ↩
                   ]
                                          [
                TLWindow
                 must have been called]

Puts up a pen select/palette requester

Call:   A4 as set by Front.i
        D0: -1 = load prefs palette
             0 = pen select only
             1 = pen select, palette select enabled
             2 = palette select only

Back:   D0=1+ being the last colour clicked, or
        D0=0  if bad, when   xxp_errn(a4) = 0 if user cancelled, else
                             xxp_errn(a4) has
                error code
                .
                             (e.g. if < version 39 of the amiga libraries)

Notes:  1. The prefs that are set are saved in ENV:/ENVARC: Tandem/Color
```

```
        2. The user is also invited to name the palette saved, & store it
           in ENV:/ENCARC: Tandem/Colors/

        3. TLReqcolor always has built-in
                Help
                        4. This routine requires release 3.0+ (V39+) of Amiga  ↩
                           libaries

        5. TLReqcolor will work ok on ECS machines

        6. This requester is not font sensitive - it always uses Topaz/8

        7. Note that tandem.library does NOT automatically open the prefs
           font. To load it, call TLreqcolor with \1 = -1. To do so,
           TLWindow need not have been called, but xxp_Screen(a4) must
           exist (e.g. if you've called TLscreen). I recommend you always
           load the prefs palette if using a private screen, but not
           necessarily if using a public screen.

MACRO:  TLreqcolor   \1=0/1/2 for D0     returns EQ if bad/cancel

Example:    [a4 as set by Front.i]    or    [a4 as set by Front.i]
            move.l xxp_tanb(a4),a6          TLreqcolor #1
            moveq #1,d0                     bne Good
            jsr _LVOTLReqcolr(a6)           tst.l xxp_errn(a4)
            tst.l d0                        beq Cancel
            bne Good                        bra Bad
            tst.l xxp_errn(a4)
            beq Cancel
            bra Bad

Sample program:  Tandem/Teaching/50.asm
```

1.249 qinf

```
                D0 = TLReqinfo(D0-D2,A4)           [
                TLWindow
                 must have been called]

Puts up an information box, with optional OK box & optional Cancel, or a
custom set of 1+ buttons. The first string in the info box is highlighted.

The requester has buitl-in help if xxp_Help(a4)=0 on call.

The user clicks a box to return, or a function key, or:
    if D1 was 1, the <Return>
    if D2 was 2, the <Return> or <Esc> keys, or <Left Amiga> with <V> or <B>.

See also
                Tandem Requesters
                .

Call:  D0=1st string
       D1=no. of strings (1+)
       D2=1 for an OK box
```

```
          2 for OK and Cancel boxes
          3 when D1 must be 2+. The last string is the text within 1 or
            more buttons, separated by \'s.

Back:  bad if D0=0, when xxp_errn(a4) has
               error code
               .
       else,  D0=1+ for choice

MACRO: TLreqinfo  \1=1st strnum  [\2=num strs [\3=1,2, or 3]]
       sets EQ if bad       defaults: \2=1  \3=1

Example:    info in strings 30-38, string 39 is   DC.B '1\2\3\4',0

            [a4 as set by Front.i]     or     [a4 as set by Front.i]
            move.l xxp_tanb(a4),a6            TLReqinfo #30,#10,#3
            moveq #30,d0                      beq Bad
            moveq #10,d1
            moveq #3,d2
            jsr _LVOTLReqinfo(a6)
            tst.l d0
            beq Bad

Sample programs:   Tandem/Teaching/39.asm   Tandem/Teaching/40.asm
```

1.250 qipt

```
               D0 = TLReqinput(D0-D2,A4)          [
               TLWindow
                must have been called]

Puts up a string requester with ok/cancel boxes

See also
               Tandem Requesters
               .

Notes:   1. User clicks Ok/presses Return for ok (or left amiga v)
         2. User clicks Cancel/presses Esc for cancel (or left amiga b)
         3. If D1=0 on call, user can type anything in box
         4. If D1=1 on call, user can type 0-9 in box
         5. If D1=2 on call, user can type 0-9 or A-F in box
         6. Typing is as per
                TLReqedit
                 (but plain text only)
         7. If xxp_Help(a4)=0 on call, there is default help built in.

Call:  A4 is set by Front.i
       The prompt is in xxp_buff   (i.e. a4)
       D0=string number of header
       D1=0 for a string,  -1 for a number,  +1 for a hex number
       D2=max number of chrs (width of box will be D2+1 ens)

Back:  buffer updated as per typing into it
       D0=0 if no input, when  xxp_errn(a4) =  0 = user selected cancel, else
```

```
                                xxp_errn(a4) has
                error code
                .
        If good, xxp_kybd+0,4,8,12(a4) = TLKeyboard D0-D3 of delimiter
        If good, and D1 was -1 or +1, the number input is in xxp_valu(a4)

MACRO: TLreqinput   \1=strnum of header [ \2=str/num/hex [ \3=max chrs]]
       sets EQ if bad/canc
       default \2= 'str'
       default \3 = #20 for string, #4 for num, #8 for hex
       puts xxp_valu in D0 (which is the number input if \2=num or hex)

Example - header = string 17, get a number 0-999, prompt '32':

            [a4 as set by Front.i]     or     [a4 as set by Front.i]
            move.b #'32',(a4)                 move.b #'32',(a4)
            clr.b 2(a4)                       clr.b 2(a4)
            move.l xxp_tanb(a4),a6            TLreqinput #17,num,#3
            moveq #17,d0                      bne.s Good
            moveq #-1,d1                      tst.l xxp_errn(a4)
            moveq #3,d1                       beq Cancel
            jsr _LVOTLReqinfo(a6)             bra Bad
            tst.l d0                         Good:
            bne.s Good
            tst.l xxp_errn(a4)
            beq Cancel
            bra Bad
           Good:
            move.l 500(a4),d0

Sample program:  Tandem/Teaching/36.asm
```

1.251 qfnt

```
                D0 = TLReqfont(A4,D0)             [
                TLWindow
                 must have been called]

Puts up a requester to select a font from those already in tandem.library's
suite of fonts. Optionally, the list of fonts can be altered by the user.

Call:    A4 as set by Front.i
         D0 = forbids         set bit 1-9 to forbid the user from altering
                              any of fonts 1-9. Font 0 can never be altered.

Return:  D0 = 0 if bad, when   xxp_errn(a4)=0 if user cancelled, else
                               xxp_errn(a4) has
                error code
                .
         else D0 = 1 to 10 = font selected + 1

MACRO:   [\1 = forbids]  default 0, i.e. all changeable except font 0.
         Returns EQ if cancelled or bad.

Example:   [A4 as set by Front.i]    or    [A4 as set by Front.i]
```

```
          move.l xxp_tanb(a4),a6          TLreqfont
          moveq #0,d0                     bne.s Good
          jsr _LVOTLReqfont(a6)           tst.l xxp_errn(a4)
          tst.l d0                        beq Cancelled
          bne.s Good                      bra Bad
          tst.l xxp_errn(a4)             Good:
          beq Cancelled                   move.w d0,font
          bra Bad
         Good:
          move.w d0,font
```

Your program can then use TLnewfont to attach the chosen font to a window. If you do not forbid a font from being altered, the user can load a font to empty slots, or replace existing fonts by other fonts.

1.252 qsho

```
               D0 = TLReqshow(d0-d2)              [
               TLWindow
                must have been called]
```

Puts up a scrolling window with information & a slider & buttons. TLReqshow is a powerful requester, which however takes some programming skill to use. It is useful since it allows the interactive display of dynamic information.

See also

```
               Tandem Requesters
               .
TLReqshow has built-in default
               Help
                if xxp_Help(a4)=0.
```

Call: A0 *must* point to a subroutine (hook) which given a line number in D0, prepares line D0 (if necessary), and points A0 to line D0. TLReqshow pushes the calling regs, then pokes their stack address in xxp_Stak(a4) for use by your hook if required. No regs (except A7) need be preserved by the hook. Lines are numbered from 0. Lines too long to display will be truncated. The line may be placed in the xxp_buff, i.e. A0 may return = A4

D0=string number of hail.
D1=total no. of strings (must be >4) (if you wanted to use fewer, set D1=5, and return nulls for unwanted lines).
D2=number of strings that can fit on requester (Must not exceed D1) (maximum 28 for PAL, 21 for NTSC, I suggest maximum 20)
D3=line num of line initially at top of display (0+)

The requester normally has 3 buttons at the bottom: Start, End and Quit. But if you set bit 31 of D2, there will be Seek forward, Seek back and Seek left buttons, enabling the user to seek particular strings within the strings displayed. TLReqshow accomplishes this by calling xxp_Hook and looking for the string sought in what is returned. You may also set bit 30 of D2, when the requester can call the hook to do a "smart" search (advanced usage).

```
       When TLReqshow calls xxp_Hook, if bit 31 of D0 is unset, then the
       hook should return with A0 pointing to the ASCII of line D0. But if
       TLReqshow calls the hook with bit 31 of D0 set, then line D0 has
       been clicked by the user. In that case, do not point A0 to ASCII
       for line D0, but instead return a code in D0 to TLReqshow as below:

       D0<0  (e.g. unchanged or -1)  do nothing
       D0=0  quit
       D0=1  redraw the requester, with no lines complemented
       D0=2  redraw the requester, complement line D0
       D0=3  redraw the requester, D1,D3 being new D1,D3 as above
             (this allows hierarchical displaying)

       TLReqshow puts the complemented line number (if any) in xxp_lcom(a4),
         so you can tell TLReqshow which line to complement by poking it
         into xxp_lcom(a4)  (in addition to the above method).
       Only 1 line can be complemented, so if you return D0=2, if another
       line had previously been complemented it will no longer be so.

       If TLReqshow is called with bit 31 and 30 of D2 set, then it means
       to do smart searching. In that case, xxp_Hook can be called with D0
       having bits 31 and 30 set, and bits 0-29 being the line to search
       from, the string sought in xxp_buff, and D1 being 3/4/5 for search
       forward/back/left. xxp_Hook must then seek the string itself, and
       return:

       D0 = the string number where found, or
       D0 = -1 = string unfound, or
       D0 = -2 = abandon smart search, do a dumb search.

       Tandem/Teaching/56.asm has a smart search.

       The requester that Tandem uses to display symbols is an example of
       advanced use of TLReqshow. See also the requester that the Multiline
       text editor puts up if you request "Print".

       Since the hook knows what line you have clicked, you could do all
       sorts of things like displaying pictures, or playing sounds, etc.,
       when a particular line is clicked.

Back:  During operation, xxp_Hook keeps getting called with
          D0 being the line num to be placed in buffer.

       bad if D0=0 when xxp_errn(a4) has
                error code
                .

MACRO: TLreqshow  \1=addr of hook subroutine
          \2=strnum of header,buttons  \3=total lines  \4=window lines
          [\5 is initial 1st line] default \5 is #0
          [\6 is 'seek' for seek buttons
                  'smart' for smart seeking by xxp_Hook] default no seek
          [\7 is initial line to complement] default none

Sample programs:  Tandem/Teaching/41.asm
                  Tandem/Teaching/49.asm
```

```
                  Tandem/Teaching/56.asm
```

1.253 data

```
                D0 = TLData(A4,D0,D1)              [
                TLWindow
                 must have been called]

Puts up a requester-like window of information, which stays there until you
call TLreqoff.

Note that unlike TLReqinfo, the user does not interact with TLdata, and
presumably your program will call TLreqoff when some event occurs.

You can NOT put up a requester (or another TLData) while TLData is showing.

Call:    A4 as set by Front.i
         D0 = string num of 1st string to be displayed.
         D1 = number of strings to display.

Return:  D0 = 0 if bad

MACRO:   TLdata  \1 = 1st string num  \2 = no. of strings  EQ if bad

Example: (the info is in strings 20 to 24)
         [A4 as set by front.i]      or      [A4 as set by Front.i]
         move.l xxp_tanb(a4),a6              TLdata #20,#5
         moveq #20,d0                        beq Outofmem
         moveq #5,d1
         jsr _LVOTLData(a6)
         tst.l d0
         beq Outofmem

Sample program:   Tandem/Teaching/55.asm
```

1.254 pgrs

```
                ()=TLProgress(D0,D1,D2,A4)         [
                TLWindow
                 must have been called]

Puts up a visual progress thermometer on the
                currently popped window
                ,
optionally with written numbers or % superimposed on it. This is strictly a
rendering routine - the thermometer is not a gadget. The thermometer can be
a one-off, or continually updated. If the thermometer doesn't fit, it will
be trimmed. The progress is always shown horizontally. Subsequent calls
simply overlay what was there before. If text or % is specified, it is always
Topaz/8. The appropriate pens in xxp_pref(a4) are used.

Call:   Before call, set xxp_prgd+0,4,8,12(a4) with xpos,ypos,width,height
```

```
        A4 as set by Front.i
        D0 = the progress
        D1 = the total.  D0 must be <= D1
        D2 = 0 no text,  -1 text in form prog/total  +1 percent

Back:   no result; xxp_errn(a4)<>0 if error, i.e. window resized.

MACRO:  \1 = the progress   \2 = the total
        [\3 = txt or %] default no text or %

Example:   [A4 as set by Front.i]    or   [A4 as set by Front.i]
           [xxp_prgd(a4) vals set]         [xxp_prgd(a4) vals set]
           move.l xxp_tanb(a4),a6          TLprogress prog,totl,%
           move.l prog,d0                  beq Resized
           move.l totl,d1
           moveq #1,d2
           jsr _LVOTLProgress(a6)
           tst.l xxp_errn(a4)
           bne Resized

Sample program:  Tandem/Teaching/54.asm
```

1.255 prfs

```
                () = TLPrefs(D0,A4)                [
                TLWindow
                 must have been called]

Puts up a requester, with lots built-in help and other features, to allow
the user to set preferences for all programs that use tandem.library. The
new preferences are put in ENV: (& of course ENVARC: if Save), where they
are available to new programs opening with tandem.library. Programs load
the preferences when TLwindow is called.

The preferences also become operative in all requesters opened under the

                currently popped window
                 when the user selects "Use" or "Save". However, they
do not become avaiable to other already opened windows, and so must be
poked into those windows "by hand" if required. The preferences set by
TLPrefs can easily be over-ridden.

When tandem.library opens a requester, if the requester won't fit, it will
progrssively automatically over-ride preferences as required to try and make
the requester fit.

Note that when TLMultiline is running in a window, the first menu always
has "GUI Preferences" in its "Project" window, so the user can set
preferences.

(Concerning the colour palette, see also
                TLReqcolor
                ).

Call:    A4 as set by Tandem.i
```

```
         D0 = +1 disable change palette   -1 enable change palette

Return:  none (xxp_errn<>0 if an error ocurred)

MACRO:

Return:  none

MACRO:   [\1 = color] enables change palette   default can't change palette

Example:    [A4 as set by Front.i]      or    [A4 as set by Front.i]
            move.l xxp_tanb(a4),a6            TLprefs color
            moveq #-1,d0
            jsr _LVOTLPrefs(a6)

Sample program:   Tandem/Teaching/68.asm
```

1.256 mlti

```
                D0-D4 = TLMultiline(A4)            [
                TLWindow
                 must have been called]

Uses the
                currently popped window
                 to edit a text buffer.

Notes about TLMultiline

TLMultiline is a large and complex routine, which along with TLReqedit takes
up the majority of tandem.library. It has extensive context sensitive online
help built in, along with an AmigaGuide file which can be viewed from its
menu called "Multiline.guide" which should be bundled up with any
program you write which calls TLMultiline.

TLMultiline co-opts a window as a sort of "gadget" in order to create and
maintain a table of ASCII lines. TLMultiline is thus a flexible text editor
for use within your program. It grabs the currently popped window and uses
it to insert, delete & edit lines of ASCII text. Line editing is as per

                TLReqedit
                 (which TLMultiline calls). If the window that
TLMultiline uses has scrollers, TLMultiline will use them.

When you open any window, it gets a 0 in xxp_Mmem to show your window is
not (yet) used for TLMultiline, and puts defaults in the other TLMultiline
data.

The first time you call TLMultiline for a window, it creates memory using
_LVOAllocVec (NOT AllocRemember), and, if you close the window, the
TLMultiline memory will be released. [If you want it unreleased, cache its
address, and put 0 in its xxp_Mmem before calling TLWsub].

Before calling TLMultiline for a window, you should poke the memory size you
want into xxp_Mmsz, else the default will be 10000 bytes. You can also
```

poke a value into xxp_Mmxc, to over-ride the default 76 characters per line.

TLMultiline returns if the user makes a menu quit selection or makes another window active, or presses <Esc>. You can then simply re-call it to continue editing. When TLMultiline resumes, it uses xxp_Mtpl as the top line visible on the window, and xxp_Mcrr as the line with the cursor. You can change these, and also the contents of memory, if you wish, since TLMultiline always checks the validity of the contents of xxp_Mtpl and xxp_Mcrr.

After TLMultiline returns, the memory at xxp_Mmem will contain a series of ASCII strings delimited by $0A characters, like ED. The address of the byte after the last line will be in xxp_Mtop, and the number of lines in memory will be in xxp_lins(a4).

Tandem's sc editor and jotter viewers are examples of TLMultiline usage.

Whatever TLMultiline does to a window - attaching a menu, fonts &c to it, will all be reversed at exit from TLMultiline - TLMultiline leaves the window just as it found it, except that you will need to clear the window and refresh its contents if any. Also, if it has scrollers, you'll need to update them.

TLMultiline's menu strip has a "Text Style" menu. This enables the user to select fonts, text spacing, text styles, &c. When TLMultiline saves a file, it also saves any such data in a file of the same name as the plaintext, with .styl appended. When TLMultiline loads a file, it will also load the .styl file if present (unless you forbid all styl data when you call TLMultiline).

TLMultiline has an "about" item in its menu strip, with dfault info. However you can over-ride this info by putting the 1st string number, and number of strings, in xxp_about, like this. Suppose you have about info in strings 14-23:

```
  move.w #14,xxp_about(a4)
  move.w #10,xxp_about+2(a4)
```

(n.b. the version of TLMultiline distributed with release 1 of Tandem does not have text style & line style features yet operative. These are still under development. Thus TLMultiline is as yet merely a plaintext editor).

```
Call:  A4 as set by Front.i
       D0 has whichever of these bits you want, OR'd together
    xxp_xmsty: equ $FFF0        ;disable all changes to styl
    xxp_xchnd: equ $0001        ;<> = text changed (on retn)(ignored on call)
    xxp_xunsv: equ $0002        ;<> = text unsaved (on call & retn)
    xxp_xpage: equ $0010        ;disable paging
    xxp_xblok: equ $0020        ;disable blocking
    xxp_xspce: equ $0040        ;disable lspace
    xxp_xspac: equ $0080        ;disable cspace
    xxp_xjust: equ $0100        ;disable fjst change
    xxp_xpens: equ $0200        ;disable pens
    xxp_xcols: equ $0400        ;disable screen colour
    xxp_xpict: equ $0800        ;disable graphics
    xxp_xfnts: equ $1000        ;disable font select
    xxp_xrend: equ $2000        ;disable text rendering
    xxp_xchng: equ $10000000    ;disable all alterations to text
```

```
      D1 has whichever of these bits you want, OR'd together
    xxp_xesty: equ $0FFF        ;disable all changes to styl
    xxp_xbold: equ $0001        ;disable  Ctrl B bold
    xxp_xital: equ $0002        ;disable  Ctrl I italic
    xxp_xundl: equ $0004        ;disable  Ctrl U underline
    xxp_xdubl: equ $0008        ;disable  Ctrl W wide
    xxp_xoutl: equ $0010        ;disable  Ctrl O dotted undeline
    xxp_xshad: equ $0020        ;disable  Ctrl S shadow
    xxp_xrjst: equ $0040        ;disable  Ctrl R right just
    xxp_xfjst: equ $0080        ;disable  Ctrl J full just
    xxp_xcent: equ $0100        ;disable  Ctrl C centre
    xxp_xljst: equ $0200        ;disable  Ctrl L left just
    xxp_xcmpj: equ $0400        ;disable  Shift Ctrl C complement
    xxp_xsusb: equ $0800        ;disable Ctrl up/down arrow super/subscript
    xxp_xunrm: equ $1000        ;disable return if menu select
    xxp_xunre: equ $2000        ;disable return if unknown Ctrl key
    xxp_xunrc: equ $4000        ;disable return if unknown other than Ctrl

Back:  xxp_errn = 0 if ok (possible errors 1,2,30,31,32,34)
       xxp_Mmem,Mtop,Mcrr,Mmxc,Mtpl set
       xxp_chnd  bit0=1 if changed,  bit1=1 if unsaved
       xxp_lins  set
       xxp_kybd  set  $1B=Esc  $93=Close  $97=Inactive

MACRO: TLmultiline    \1,\2 as per D0,D1 above. Returns EQ if bad

See also:
               TLattach
               Example:  see Teaching/44.asm
```

1.257 tlat

```
               TLattach  MACRO only              [
               TLWindow
                must have been called]

Normally,
               TLMultiline
                creates its own memory, the first time
TLMultiline is called for a window. However, you may have reason to want to
create memory "by hand", and then attach it to a window for use by
TLMultiline. To do that:

1. create the memory. You *must* use _LVOAllocVec.
2. call TLattach after the window is created, but before you have called
   TLMultiline for that window.
3. You need never free the memory - Front.i will do so at closedown time.
   But, if you wish to free it, then:
    (a) _LVOFreeVec the memory
    (b) poke zero into xxp_Mmem of that window in xxp_WSuite
4. You can then re-start TLMultiline with that window, if you want to.

MACRO:   TLattach
         A4 as set by Front.i
```

```
         A5 the window's xxp_WSuite entry
         \1 the memory address
         \2 the memory size
```

1.258 help

```
                Context Sensitive Help             [
                TLWindow
                 must have been called]
```

In the xxp_tndm structure, you will see:

```
    xxp_Help LONG
```

You can poke a reference to a series of strings, as follows:

```
    * poke the stringnum of the first.W to xxp_Help
    * poke the stringnum of the number of strings.W to xxp_Help+2
```

e.g. suppose the help is in strings 132-136 (5 strings inclusive). Then you would poke:

```
    move.w #132,xxp_Help(a4)
    move.w #5,xxp_Help+2(a4)
```

Then, whenever you call TLkeyboard, if the user presses the <Help> key it will bring up a requester displaying the help. Then, when the user presses the OK button on the help, it will go back to what it was doing before. This works, even if the user was inputting to one of the requesters, TLReqchoose, for example. This makes it easy to provide context sensitive help. All of the requesters (TLReqinput, TLReqcolor, TLReqshow & TLMultiline etc.) have default help built in, which is shown if xxp_Help is zero.

To switch off the Help key, just poke 0 into xxp_Help:

```
    clr.l xxp_Help(a4)
```

Attaching an AmigaGuide to the Help Requester

If you do the following:

1. set bit 7 of xxp_Help+2(a4)
2. put the address of the filepath of the guide relative to the CD in xxp_guid(a4)
3. put the address of a string with the node name to open at in xxp_node(a4)

```
Example:    move.w #132,xxp_Help(a4)
            move.w #5,xxp_Help+2(a4)
            bset #7,xxp_Help+2(a4)
            move.l #guidename,xxp_guid(a4)   ;guidename: dc.b 'Fred.guide',0
            move.l #nodename,xxp_node(a4)    ;nodename: dc.b 'whatever',0
```

Sample Program: Tandem/Teaching/70.asm

1.259 prim

```
                tandem.library primitive routines

tandem.library has the following primitive routines:

TLreqedit       ;edit a string
                TLReqedit
                TLpassword      ;enter a password
                TLPassword

                Button Routines
                TLButstr        ;calculate button sizes
                TLButstr
                TLButprt        ;render buttons
                TLButprt
                TLButtxt        ;text in buttons
                TLButtxt
                TLButmon        ;monitor buttons
                TLButmon

                Slider Routines
                TLSlider        ;render slider
                TLSlider
                TLSlimon        ;monitor slider
                TLSlimon

                Tabs Routines
                TLTabs          ;render tabs
                TLTabs
                TLTabmon        ;monitor tabs
                TLTabmon
                TLPict          ;draw little picture
                TLPict

                Drop Down Menu
                TLDropdown      ;draw/monitor a drop down menu
                TLDropdown

                Custom Requesters
                TLReqredi       ;prepare for requester
                TLReqredi
                TLReqchek       ;requester dims
                TLReqchek
                TLReqon         ;render requester
                TLReqon
                TLReqoff        ;requester off
                TLReqoff
                Primitive Routines

Here, in a general way, is what you can do with primitive routines:

1. TLReqon and TLReqoff allow you to put up a custom requester, like
   TLReqchoose, TLReqinfo, &c to your own design.

2. TLWupdate and TLWcheck allow you to plan your work around resizable
```

windows. See also
Window Refreshing
.

3. The remainder allow you to have arrays of buttons, sliders and editable text strings etc. on your window, which can be simply monitored by TLKeyboard, in a way that is easier (but less sophisticated) than gadtools or BOOPSI objects. These, together with TLReqmenu, allow you a comprehensive user interface, fully in accord with Amiga programming guidelines. (Once you get lots of experience with these, you can graduate to boopsi objects, datatypes & the like).

1.260 qedt

```
() = TLReqedit(D0-D7,A4)          [
TLWindow
 must have been called]
```

Edits or Displays a string

About TLReqedit

TLReqedit is an enormously powerful and complex routine for the display or editing of a string. TLReqedit uses tags, the only TL routine to do so. In the 1024 byte block of data pointed to by A4, you will see a pointer labelled as xxp_FWork. This contains a block of memory which TLReqedit uses for its internal workings. The first thing you decide about TLReqedit, is what task you want it to do. The types of tasks are as follows:

1. Calculate the width of a string, without displaying it.
2. Display a string (but don't edit it).
3. Display a text string for editing.
4. Display a decimal number string for editing.
5. Display a hexadecimal number string for editing.
6. Display a text string for editing, which is one of a series of strings in a memory buffer - this string can be split into 2 lines by pressing <RETURN> or by its overflowing past its maximum length.

The calling program specifies a "tablet" for the string, i.e. the top left pixel coordinates of where it is to be displayed, and the width of the tablet in pixels. If the string is or becomes too long to fit in its tablet, it will scroll sideways to keep the cursor visible.

Text strings can be displayed in plaintext, or with many different character or line formats. Here is a list of possible options. For explanations of the options, see the docs:

1. characters can be plain, bold, italic, drop-shadow (or any combination) (drop shadow x and y offsets & pen can be specified).
2. characters can be normal width or double width.
3. characters can be superscript or subscript.
4. characters can have the following underline styles:
   - single underline
   - dotted underline
   - strike through

```
    - overline
    - double underline
    - underline + overline
    - double underline + overline

 (Note that all possible combinations of the above can be shown character
  by character).

 5. text can be normal or complemented.
 6. foreground, background, and drop-shadow pens can be specified.
 7. character spacing can be specified.
 8. line justification be:
    - normal (left justified)
    - right justified
    - centered
    - full justified (i.e. spread across the line like professional printing)
 9. printing is normally jam2, but jam1 if required (e.g. to overlay a
    background).
 10. the calling program can force proportional fonts to print fixed (e.g.
    for formatting tables) if required.
 11. character case can be:
    - normal
    - all upper case
    - all lower case
    - small caps
 12. characters can be displayed left to right (normal), or right to left.
 13. the tablet can be cleared before display.
 14. various keyboard combinations and menu options make things easy for the
    user, and TLReqedit displays built-in help automatically unless the
    programmer over-rides it.

You will see that the above allow all aspects of internet cascading style
sheet printing, and then some.

TLReqedit Tags

There are many tags for TLReqedit. You must pay careful attention to the
instructions below for using them. None of the tags is compulsory.

1. xxp_xtext
   Specifies the address of the text. The default is (A4) (i.e. xxp_buff).
   If the cursor is off (i.e. display only) TLReqedit prints the text from
   wherever xxp_xtext appears. But if the user is editing it, then
   TLReqedit caches it in the first 256 bytes of xxp_FWork. You can put
   the text to be edited into xxp_FWork before calling TLReqedit if you want,
   and point xxp_xtext there, which saves time. If you do use (A4) for the
   input text, then TLReqedit will copy the text as edited back there before
   it returns. The text must be null delimited.

2. xxp_xstyl
   Specifies the address of character styl data. The default is that all
   character styl data is 0 (i.e. all characters are plain). See also the
   xxp_xstyb tag below. If the text is to be edited, TLReqedit will copy or
   create styl data in xxp_FWork+256. You can find the final state of the
   styl data in xxp_FWork+256 after TLReqedit returns. Of course, if styl
   data is forbidden (see xxp_xforb below), then it will be all 0's, and
   of no interest. There is 1 styl byte for each text byte, including the
```

text's null delimiter. styl does not have a null delimiter. styl is documented in tandem/support/tanlib.i, as follows:

```
(a) bit 0    bold
(b) bit 1    italic
(c) bit 2-5  0001 underline          0110 double underline
             0010 superscript        0111 double underline + overline
             0011 overline           1000 dotted underline
             0100 subscript          1001 strike through
             0101 under + overline
    (some of these need reasonably-sized text to display meaningfully)
(d) bit 6    shadow font
(e) bit 7    double width font
```

So, for example, the styl for a character which is plaintext would be 0, while the styl for a character which is italic would be 2. If you are storing the styl for many lines, then then CmpByteRun is perhaps the best compression method.

If you make, say, A5 point to xxp_FWork like this:

```
move.l xxp_FWork(a4),a5
```

then the 256 bytes from (A5) will hold the null-delimited ascii, while the styl will be the corresponding 256 bytes in 256(A5).

3. xxp_xmaxt
   Specify the width of the tablet with xxp_xmax. The default is to extend it to the rightmost pixel of the window (excluding the border). The height of the tablet is always the font height. If the user sizes the window so it is narrower than the tablet width, TLReqedit will use whatever of the tablet is available. (See below about how TLReqedit handles window sizing). But if the user specifies xxp_maxw, then that will also be the default for xxp_maxt, unless it won't fit.

4. xxp_xmaxc
   Specifies the maximum number of characters the text can have. Default is 254, and xxp_xmaxc cannot exceed 254. If the initial text exceeds 254 or xxp_xmax, it will be truncated.

5. xxp_xmaxw
   Specifies the maximum pixel width of the text. The default is there is no limit (i.e. xxp_maxc limits the length). If both xxp_maxc and xxp_xmaxw are specified, TLReqedit will not allow either to be exceeded. Of course, xxp_xmaxw can be wider than xxp_xmaxt, in which case the text will automatically scroll sideways to make the cursor visible (but see also under xxp_xoffs). If the initial text exceeds this limit, it will be truncated.

6. xxp_xcrsr
   Specifies the cursor position, relative to 0 being the first character. The default is 0. If you specify a value beyond the last character, then it will be placed just after the last character (or on the last character if xxp_xmaxc has been reached).

   If you specify -1 for the cursor, there will be no cursor. TLMultiline will display the text, and return immediately. That is to say, no editing

of the text takes place. In such cases, TLReqedit does not necessarily copy the text to xxp_FWork before printing it.

7. xxp_xoffs
   Specifies a fixed offset. If text for display only is too long to display in its entirety, TLReqedit prints whatever will fit on the window within the tablet. But if you specify an offset of say 200 pixels, then the left of the tablet will show all text from the 200th pixel onwards, if any. You might, for example, be monitoring a scroller, and showing portions of the text according to the position of the scroller.

   If you are editing text, then TLReqedit will normally automatically offset text to keep the cursor visible, so you won't need xxp_xoffs. But if the line you are editing is one of several in a buffer, you may want to offset all of them in step. In that case, you would specify an initial fixed offset compatible with where the cursor is (see how to find the cursor position under xxp_xnprt below). Then, if the user's typing &c puts the cursor outside the tablet, but still on the line, TLReqedit returns with D0=10 (i.e. bad fixed offset); you can then re-display all the lines with an appropriate fixed offset, and re-edit the line. You can thus-wise move the lines across in steps of say 100 or 200 pixels at a time.

   If you supply -1 for xxp_xoffs, there is no fixed offset, and that is the default.

8. xxp_xforb
   Allows you to forbid various features of TLReqedit.
   Tandem/support/tanlib.i has the bits to set to forbid various options - e.g. xxp_xbold forbids bold font. If you use xxp_xesty for xxp_xforb, then everything is forbidden, and you can only edit plaintext. Note however that whatever styl you start with is allowed - it is the act of typing Ctrl/b which switches bold text on and off that's forbidden by xxp_xbold, not the display of bold text which might already exist due to the initial styl when TLReqedit was called. The default for xxp_xforb is xxp_xesty, i.e. everything is forbidden by default, so if you wish the user to be able to specify character style, line format, &c then you have to use xxp_xforb with those things NOT forbidden.

9. xxp_xtask
   The task values you specify with xxp_xtask are as follows:

   0 string - no continuation line
   1 string - continuation line
   2 number - decimal format
   3 number - hexadecimal format

   The default is 0 (string - no continuation line).

   To input a string, or to simply display a string, task=0 (the default). If you specify 2 or 3, TLReqedit allows you to type a decimal number or a hex number only; on return, the value will be in xxp_valu, so you can get it e.g. into D0 by MOVE.L xxp_valu(A4),D0
   xxp_valu is a longword, so that is the limit of values that TLReqedit can input, i.e. unsigned values from 0 to $FFFFFFFF. For a float, input a string, & use TLFloat to get its value.

   If you are editing a series of lines in a buffer, use xxp_xtask=1...

When you are typing a set of lines into a buffer, then if you press <return>, or push characters forward until the line fills up, TLReqedit will split your line into two lines and return. The original line, as edited, will be in xxp_FWork+0, and its styl in xxp_FWork+256. The continuation line will be in xxp_FWork+512, and its styl in xxp_FWork+768. If TLReqedit returns with a continuation line existing, then the longword xxp_chnd(A4) will have bit 15 set (i.e. the byte xxp_chnd+2(A4) will be negative). The cursor then can be either in the original line or the continuation line, and if bit 14 of xxp_chnd(A4) is set (i.e. bit 6 of the byte xxp_chnd+2(A4) is set), then the cursor will be in the continuation line, else it will be in the original line.

The business of typing a series of lines into a buffer is all taken care of for you by TLMultiline of course, so all you need to do is use that, and you can forget all about continuationm lines.

10. xxp_xcomp
If you send -1 to xxp_xcomp, it will be sent complemented, or if you send 0 it will not be complemented. The default is 0. The user can also press Shift/Ctrl/C to switch complement on and off, unless xxp_xforb forbids it. See also TLReqedit return codes below.

11. xxp_xnprt
If you send -1 to xxp_xnprt, TLReqedit will calculate the width of your text, and the position and width of its cursor, and return without printing. If you send 0 to xxp_xnprt, then TLReqedit will go ahead and print the string. The default is 0. Using xxp_xnprt allows you to measure the printed length of strings. Note that styl bytes can affect the width of strings, as can several of the parameters, so you should always supply the same tags with xxp_xnprt as you will without it, or it may not return the correct value.

12. xxp_xfont
Specifies the font number (which must have had TLGetfont or TLAslfont called for it). You can also use TLReqfont to allow the user to choose a font. The default is the font already attached to the xxp_wsuw of the

currently popped window
. If you specify xxp_xfont, then the font which was attached to the window before you called TLReqedit will be re-attached to it when TLReqedit returns.

Note that when you vary the font style with styl bytes, TLReqedit uses Amiga graphics.library calls to vary the font style. If you request superscript or subscript, TLReqedit uses diskfont.library to load a smaller version of the font into memory. If you request double width, There will be a small delay the first time you do so for a particular font, as TLReqedit uses the normal width font to construct a double width font, save it to disk as "FONTS:Temporary.font", and then load it into memory.

13. xxp_xcspc
Specifies the pixels between characters - the default is 0. The value sent to xxp_xcspc can be from 0 to 31.

14. xxp_xmaxj
If you specify a line justification of "full justify", then TLReqedit

spaces out the characters in the line until the lines exactly fit the xxp_xmaxw you specify. This is one of the really nifty abilities of TLReqedit. xxp_xmaxj specifies the maximum blank pixels to put between characters to space them out. The default value is 5. A low value means that characters won't look ugly by being spaced out too far, but if you specify a value too low, then you might get an unjustified line in the middle of a paragraph, which can be ugly. If you specify a value of 0 for xxp_xmaxj, then full justification is forced, even if there are only 2 characters in the line. Otherwise, you can specify 1 to 15.

15. xxp_xltyp
You can specify the following values for xxp_xltyp:

0 left justify (the default)
1 center
2 right justify
3 full justify

TLReqedit's specialty is full justify, which spreads the characters right out over the line, like in professional typesetting. Unless you forbid it, the user can also select the line justification. See also xxp_xmaxj above.

16. xxp_xkybd
Allows you to specify the xpos of the cursor. If the user has clicked the line, causing you to call TLReqedit to edit it, you might want to put the cursor on whatever character the user clicked. In that case, subtract the left hand value of the tablet relative to the left of the window, and send that value to xxp_xkybd. This value will over-rule xxp_xcrsr, unless xxp_crsr is -1, when xxp_xkybd would be ignored.

17. xxp_xiclr
Set to -1 to cause TLReqedit to fill the tablet with the background pen before displaying the text. Else, send 0. The default is 0.

18. xxp_xtral
Set -1 to cause TLReqedit to remove trailing spaces from your input before returning. But if your only input is just a single space, it will not be removed. Else, send 0. The default is 0.

19. xxp_xshdv
Allows you to set parameters for shadow font styl. You should send a long word, as follows:

byte 0 always 0
byte 1 the shadow font pen, 0-255.
byte 2 the y offset (typically 1).
byte 3 the x offset (typically 2).

It may work to use negative offst values, I give no promises. The x offset can be 0-6, the y offset can be 0-3. The default value is as per xxp_shad in the curently popped window's xxp_shad.

20. xxp_xresz
Set to -1 to allow TLReqedit to continue if the user resizes the window while editing. Set to 0 to make TLReqedit always return if it finds the user has resized the window. The default is 0.

21. xxp_xmenu
This is a rather strange but useful tag. It is the menu number of a menu for TLReqedit. In the program tandem/teaching/60.asm you will see there the menu TLReqedit MUST have, IF it has one at all. You must have the menu items EXACTLY as set out in that program. You will see that menu also in TLMultiline. In Teaching/60.asm, the menu is menu 0, so xxp_xmenu sends 0, whereas in TLMultiline the TLReqedit is menu 3, so TLMultiline sends 3 to xxp_xmenu. The default is of course that no menu for TLReqedit is attached. The value of creating a menu for TLReqedit is that the user can use it. If you have forbidden everything and are simply editing as plaintext, there is no point in attaching a menu. It is the responsibility of the calling program to enable appropriate menu items & sub-items before calling, and disable them on return. Even if there is no menu, the user can use teh keyboard byapsses, unless they are forbidden.

22. xxp_xfgbg
This allows you to set text pens, in a longword as follows:

byte 0 always 0
byte 1 always 0
byte 2 the foreground pen (0-255)
byte 3 the background pen (0-255)

The default are the pens in the xxp_wsuw of the
currently popped window
.

23. xxp_xffix
If you set this to -1, then if the font attached is proportional, TLReqedit will display it as if it were fixed. (Like <PRE> in HTML). If the font is already fixed, it has no effect. This can be useful in setting out tables, &c. TLReqedit displays a line much slower if xxp_ffix is -1. Else, set xxp_xffix to 0. The default is 0.

24. xxp_xjam1
If you set this to -1, TLReqedit will print to the window using jam1. This can be useful to overlay a picture with the text. But note:
(a) Do not use this, unless xxp_xcrsr = -1 (i.e. display only). If you use this when editing, strange things will happen.
(b) Do not use this, if the line to be printed does not fit completely in its tablet. Else again strange things might happen.
Else, set xxp_xjam1 to 0. The default is 0.

25. xxp_xcase
xxp_xcase can be set as follows:
0 print normally
1 print all upper case
2 print all lower case
3 print with small caps
The default is of course 0. You should only use this if xxp_xcrsr is -1, since it only acts on the initial state of the display. Small caps uses subscript sized text, so it only works well for fairly large text.

26. xxp_xstyb
If you want all styl bytes to be the same (e.g. 2 for all italics), then

use xxp_xstyb. You can use this, even if you have forbidden everything. If you have also specified xxp_xstip, then xxp_styb has no effect. The default value is 0.

27. xxp_xrevs
xxp_xrevs if followed by -1 causes right to left printing, or 0 for normal left to right (the default). If you are using right to left, the text width must NOT exceed the tablet width - the text cannot scroll sideways, and also cannot be offset. But in general the tags &c work ok with xxp_xrevs. However I do not guarantee that all possible combinations of tags will work properly with xxp_xrevs. You would generally use it together with right justification, and say a Hebrew or Arabic font.

TLReqedit Return Codes
TLReqedit can return with the following return codes in D0:

0	User pressed <return>
1	User pressed <Esc>
2	Caller had xxp_xnprt = -1 (i.e. don't print)
3	Caller had xxp_xcrsr = -1 (i.e. no cursor)
4	Continuation line created (can also return with 0 when that happens)
5	User clicked left mouse button off the tablet
6	User made a menu selection not in TLReqedit's menu
7	User made an unrecognised keyboard selection
8	The window/tablet was too narrow to print any characters
9	The window height was too small to print any characters
10	The cursor moved outside the fixed offset limits (see xxp_xoffs)
11	Can't attach the specified font (out of memory?)
12	Window resized (and xxp_xresz not set to -1)
13	User pressed close window gadget
14	Window became inactive

TLReqedit ignores an active window message. If the window resizes, it will refresh only itself but nothing else, so usually you will set xxp_xresz to 0 (the default), so you can refresh the window before re-calling TLReqedit.

If you want to re-call TLReqedit, e.g. if you got, say, a return code of 7 and have acted upon it, then you would re-call TLReqedit with the same tags, but without xxp_xtext or xxp_xstip, since they'll already be in TLFwork. There is no problem however if you re-call it with quite different tags.

TLReqedit also returns the following data:

(a) in xxp_chnd(a4) if xxp_xnprt was 0:

bit 31	set if text/styl changed from it initial state
bit 16-30	the offset at the time of return
bit 15	set if there is a continuation line
bit 14	set if the cursor is in the continuation line
bit 13	set if the line was complemented on return
bit 12	set if the return code in D0 was 8-11 (error conditions)
bit 11	set if xxp_ffix forced a proportional fotn to be fixed
bit 10	set if the user changed the line justification
bit 8-9	line justification on return (0-3 as pre xxp_xltyp)
bit 0-7	unused

```
    in xxp_chnd(a4) if xxp_xnprt was -1:
    bit 0-15   the pixel posn of the lhs of the cursor (unless xxp_xcrsr=-1)
    bit 16-31  the pixel posn of the rhs of the cursor (unless xxp_xcrsr=-1)

(b) in xxp_crsr(a4)
    the final cursor position (unless xxp_xcrsr was -1)

(c) in xxp_valu(a4)
    the value input by the user, if the xxp_xtask was 2 or 3, and
    xxp_xnprt was not -1.

    If xxp_xnprt was -1, then xxp_valu:
    bits 0-15   characters in text
    bits 16-31  text width in pixels
                (caution: when a cursor in in the text, its width can
                 vary slightly from when there is no cursor).

(d) xxp_kybd has 4 longwords, being D0-D3 of the last TLKeyboard call
    (if any) before TLReqedit returned.

(e) xxp_FWork+0    has the final state of the text
    xxp_FWork+256  has the final state of the styl
    xxp_FWork+512  has the continuation line text (if any)
    xxp_FWork+768  has the continuation line styl (if any)
    If xxp_xtext pointed to (A4), then the final text will also be in (A4).

TLReqedit Further Docs

TLReqedit displays/edits the string in the
                currently popped window
                .
See also
                Primtive Routines
                Keyboard behaviour:
       Del deletes a character
       Backspace moves the cursor left and deletes a character
       Shift/Del deletes all characters from the cursor forward
       Shift/Backspace deletes all characters left of the cursor
       The Tab key jumps to tab posns & space fills
       Alt with a typeable character gives ASCII as per the prefs keymap
       Left Amiga with a typeable character adds $80 to its ASCII value
       Shift/Ctrl/U is an UNDO key - it reverses the effect of the
            previous keystroke (even if it was Ctrl/q)
       Shift/Ctrl/R is a RESTORE key - it restores the text to its aboriginal
            condition
       Shift/Ctrl/C turns COMPLEMENT on/off
       Ctrl/x erases all text
       Shift/Ctrl/S moves all text from the cursor to the end of the line
            by space filling
       A click within the tablet causes the cursor to move where clicked
            (or as far right as possible if not yet that many characters)
       A click outside the tablet causes TLReqedit to return
       All other non-typeable inputs cause Reqedit to exit

Special keyboard behaviour:
       Ctrl/l   left justify
```

```
       Ctrl/r   right justify
       Ctrl/j   full justify
       Ctrl/c   centre
       Ctrl/b   bold
       Ctrl/i   italic
       Ctrl/u   underlined (other underlines available)
       Ctrl/up arrow   superscript
       Ctrl/down arrow subscript
       Ctrl/w   double width

Call:  A4 as set by Front.i
       D0,D1  = tablet position
       A0     = tags


Back:  Return code in D0 (see above)

MACRO: TLreqedit    \1 = xpos  \2 = ypos  \3 = tags
       (or \3 = 0/1 for default tags - see tandem/jottings/refsheet)

Example:  1. Tandem/Teaching/60.asm
          2. see the use of TLReqedit in the sc for TLReqinput in Tandem.i
```

1.261 pass

```
               ()=TLPassword(D0,A4)               [
               TLWindow
                must have been called]

Puts up a little window to type in a password.
Of course, just blobs appear on the window, not the characters typed. Thus,
  someone looking over the shoulder of the person typing would not see
  the password on the window.

Call:  A4 as set by Front.i
       D0 = max chrs in password

Back:  all regs saved
       password in buff (null delimited)
       if does NOT encrypt the password in (a4) - the password will not
           exist anywhere else in memory at any time except in (A4)

MACRO: \1 = max chrs in password

Example:  [A4 as set by Front.i]     or     [A4 as set by Front.i]
          move.l xxp_tanb(a4),a6             TLpassword #6
          moveq #6,d0
          jsr _LVOTLPassword(a4)

Sample program:    Tandem/Teaching/71.asm
```

1.262 butz

```
                Button Routines

TLButstr        ;calculate button sizes
                TLButstr
                TLButprt        ;render buttons
                TLButprt
                TLButtxt        ;text in buttons
                TLButtxt
                TLButmon        ;monitor buttons
                TLButmon
                tandem.library has several routines which enable you to put a 2- ↩
                   dimensional
array of buttons on a window, and use them as buttons to be clicked. You
must act quickly and obviously on the clicking of a button, or else invert
it "by hand", since tandem.library does not invert the buttons when you
click them.

1. The button routines use the following items in the xxp_tndm structure
   at (a4) (see Tandem/Support/tanlib.i for the xxp_tndm structure):

   xxp_butx   the top left posn of the array
   xxp_buty
   xxp_butw   the width and height of each button
   xxp_buth
   xxp_btdx   the horizontal distance between button lhs (s/be >= xxp_butw)
   xxp_btdy   the vertical distance between button tops (s/be >= xxp_buth)
   xxp_butk   the no. of buttons in each row
   xxp_butl   the no. of buttons in each column

   (the number of buttons is xxp_butk * xxp_butl)

2. If you have strings to be printed in the buttons, you can join the
   strings together separated by \'s, and null delimited. Then call
   TLButstr, which will arrange the above into a horizontal row. You can
   then re-arrange the buttons into a column, or several rows.

3. Having set the above xxp_but's, you can call TLButprt to draw all the
   buttons on the window

4. If you have text as per 2, then call TLButtxt to put teh text in the
   buttons.

5. If TLKeyboard has returned D0=$80, i.e. a click, you can call TLButmon,
   which returns 0 if no button was clicked, or else 1+ being the number of
   the button clicked (This is unlike radio buttons in gadtools which
   returns IDCMP's).
```

1.263 buts

```
                ()=TLButstr(A0,A4)                  [
                TLWindow
                 must have been called]

Sets up a row of buttons, so text will fit in them
```

```
See also
                Button Routines
                Call:   A5 points to an IntuiText (e.g. an xxp_WSuite entry)
        A0 is the string of button contents

Back:   all regs saved
        xxp_butx thru xxp_butl are set up for call TLButprt & TLButtxt

Notes:  1. The string has 1 or more sub-strings, separated by \ characters,
           the whole null delimited.
        2. xxp_butx will be set up as a row of buttons, all touching each
           other, all just wide enough for the widest of the strings.
        3. You can then re-arrange the buttons, e.g. into a column by
           swapping xxp_butk & xxp_butl, or into several rows, &c.
        4. xxp_butx & xxp_buty will be 0, so adjust them to be where you
           want them.
        5. xxp_btdx will = xxt_butw, & xxp_btdy will = xxp_buth, so increase
           xxp_btdx (& xxp_btdy if you re-arrange) to spread them out
           if desired.

MACRO:  none

Example:  See
                TLButmon
```

1.264 butp

```
                D0=TLButprt(A4)                     [
                TLWindow
                 must have been called]

Draws buttons onto the
                currently popped window
                .

See also
                Button Routines

                Primitive Routines
                Call:   A4 as set by Front.i

Back:   all regs saved
        xxp_errn(a4) = 0 if drawn ok
        xxp_errn(a4) =+1 if un/partly drawn, resized before/while drawing
        xxp_errn(a4) =-1 if un/partly drawn, since not all fitted on window

MACRO:  none

Notes:  The 2 dimensional array of buttons in xxp_butx thru xxp_buty is
        drawn to the window. Any that won't fully fit are undrawn.

Example:  See
                TLButmon
```

1.265 butt

```
                D0=TLButtxt(A0)                    [
                TLWindow
                 must have been called]

Draws text into buttons on rhe
                currently popped window
                .

See also
                Button Routines

                Primitive Routines
                Call:   A4 as set by Front.i
        A0 is the string

Back:   xxp_errn(a4) = 0 if drawn ok
        xxp_errn(a4) =+1 if un/partly drawn, resized before/while drawing
        xxp_errn(a4) =-1 if un/partly drawn, since not all fitted on window

MACRO:  none

Notes:  1. Call after the 2 dimensional array of buttons in xxp_butx thru
           xxp_buty is already drawn by
                TLButprt
                        2. the array of buttons to suit the string can be set up  ↩
                           by

                TLButstr
                , which see for the string format.

Example:  See
                TLButmon
```

1.266 butm

```
                D0=TLButmon(D2,D3,A0)              [
                TLWindow
                 must have been called]

Sees if buttons in an array have been clicked

See also
                Button Routines

                Primitive Routines
                Call:   A4 as set by Front.i
        D2,D3 are as returned by a call to
                TLKeyboard
                 which
            Also returned D0=$80

Back:   D0=0  if none of the buttons was clicked
```

```
        D0=1+ being the number of the button clicked
        [xxp_errn(a4)=+1 if window resized]

MACRO:  none

Note:   Call after the 2 dimensional array of buttons in xxp_butx thru
        xxp_buty is already drawn to the window by
                TLButprt
                Example:  string 22 is 'Fred\Jack\Sue\Caroline',0
          string 21 is 'Click a button',0
          (this example doesn't care if some of the buttons don't fit on
           the window)

          [a4 as set by Front.i]
          TLKeyboard has returned D0=$80, D2,D3 are unchanged

         Draw:                      ;** get D0=1 to 4 as per string 22
          TLwupdate                 ;update window size
          TLnewfont #0,#0,#0        ;attach Topaz/8
          TLwcheck                  ;recycle is size changed
          bne Draw
          TLstring #21,#0,#0        ;print string 21
          move.l xxp_tanb(a4),a6
          moveq #22,d0
          jsr _LVOTLStra0(a6)       ;point a0 at string 22
          move.l xxp_AcWind(a4),a5  ;use currently popped window
          jsr _LVOTLButstr(a6)      ;set button dimensions
          move.l #20,xxp_butx(a4)   ;set button array top left
          move.l #10,xxp_buty(a4)
          move.l #2,xxp_butk(a4)    ;arrange buttons into 2 X 2
          move.l #2,xxp_butk(a4)
          TLButprt                  ;print the buttons
          tst.l d0
          bgt Draw                  ;recycle if window resized
          jsr _LVOTLButtxt(a6)      ;put text in buttons (a0 still str22)
          tst.l d0
          bgt Draw                  ;recycle if window resized
         Wait:
          TLwcheck                  ;recycle if window resized
          bne Draw
          TLkeyboard                ;get keyboard
          cmp.b #$80,d0
          bne Wait                  ;keep wainting until window clicked
          jsr _LVOTLButmon(a6)      ;any of the buttons clicked?
          tst.l d0
          beq Wait                  ;no, keep waiting until a button clicked
          TLreqcls                  ;clear the window
          rts                       ;D0=1 to 4 as per string 22
```

1.267 sliz

```
                Slider Routines

TLSlider        ;render slider
                TLSlider
```

```
                TLSlimon        ;monitor slider
                TLSlimon
                (note: tandem.library can also deal with window scrollers - see

                TLWscroll
                ).
```

tandem.library has several routines which enable you to put up 1 or more sliders on a window, and use them easily. You need may also attach a subroutine which acts on the slide position dynamically as it is being slid (e.g. by putting a value in a box).

1. First, you set up the physical dims of the slider in:

   xxp_slix the lhs
   xxp_sliy the top
   xxp_sliw the width
   xxp_slih the height
   xxp_totl the total value of the slider
   xxp_strs the value represented by the slide
   xxp_tops the value represented by the top/left of teh slide

   (e.g. if there are 256 things, and the slide represents 20 of them, then totl=256, strs=20, and totl can be from 0 to 236).

   The larger of xxp_sliw or xxp_slih tells whether it slider horizontally or vertically.

   12 is the minimum width for vertical sliders
   36 is the minimum height for vertical sliders

   60 is the minimum width for horizontal sliders
   10 is the minimum height for horizontal sliders

3. You can leave xxp_hook(a4)=0; or else, you can poke the address of a subroutine into xxp_hook(a4). If you to, that subroutine will be called every time the slide moves (i.e. every time you or TLSlimon).

4. TLSlider and TLSlimon push all your regs ito the stack when you call them, and then put their stack address in xxp_Stak(a4), so your hook can inspect them if required. your hook is called with all regs trashed except A4, and you can return with all regs trashed. Preumably, the first thing your hook does is inspects xxp_tops(a4).

5. Call TLSlider to draw the slider. If you ever poke new values into xxp_tops, xxp_totl or xxp_strs, call TLSlider to re-draw the slider.

6. If whenever you call TLKeyboard, it returns $80 (being a left mouse button), you should call TLSlimon. This does the following:

   (a) returns immediately if the click was not in the slider's buttons or slide, with D0=0.

   (b) adjusts xxp_tops until the user releases the left mouse button, or moves the pointer well away from the slider. The user can also click the single step buttons in the slider, which TLSlimon also

```
       detects. If TLSlimon detects a change in xxp_tops, it adjusts the
       slide position.

   (c) whenever TLSlimon changes xxp_tops, it calls your hook (if any).

   (b) finally, TLSlimon returns with D0<>0.

7. You can have an array of same-sized sliders very easily. e.g. if they
   are horizontal (like in TLReqcolor), you set everything up, and
   simply call everything 3 times with 3 different values poked in xxp_sliy.

8. You can use TLReqedit, TLSlimon and TLButmon together to process keyboard
   clicks. You just try all of them, until you find which (if any) the
   click was on top of (This is simpler but less sophisticated than
   gadtools.library, which sends IDCMP's specific to each gadget that gets
   clicked).

If the window is too small for the slider to fit, it gets clipped. If the
window is resized, TLSlider & TLSlimon return immediately with an error
code in xxp_errn(a4).
```

1.268 slie

```
               D0=TLSlider(A4)                    [
               TLWindow
                must have been called]

draws a slider on the
               currently popped window
               .

see also
               Primitive Routines

               Slider Routines
               Call:   A4 as set by Front.i
        slider dims in xxp_slix thru xxp_slih

Back:   all registers saved
        reports:  xxp_errn(a4) = 0 ok
                  xxp_errn(a4) =+1 undrawn, since window resized
                  xxp_errn(a4) =-1 undrawn, since window too small

MACRO:  none

Example:  see
               TLSlimon
```

1.269 slim

```
               D0=TLSlimon(A4)                    [
               TLWindow
```

```
                must have been called]

Sees if TLKeyboard output is in slider, & follows it if so.

see also
                Primitive Routines

                Slider Routines
                Call:   A4 as set by Front.i
        D2,D3 as returned by TLKeyboard (& D0 was $80)

Back:   returns:   D0=-1 ok. xxp_tops(a4) updated, slider redrawn
                   D0= 0 and xxp_errn(a4)= 0 -  slider was never clicked
                   D0=+1 and xxp_errn(a4)=+1 -  aborted since window resized

MACRO:  none

Example: get a value from 0 to 255 for sought

           [a4 as set by Front.i]
            string 46 is 'adjust sought; press Esc when ok',0

          Getval:                    ;** update value of sought
           TLwupdate                 ;update window size
           TLnewfont #0,#0,#0        ;attach Topaz/8
           TLstring #46,#0,#0        ;print string 46
           bgt Getval                ;go if window resized
           move.l xxp_tanb(a4),a6
           move.l #0,xxp_slix(a4)    ;slider dims
           move.l #10,xxp_sliy(a4)
           move.l #500,xxp_sliw(a4)
           move.l #12,xxp_slih(a4)
           move.l #256,xxp_totl(a4)  ;possible values 0-255
           move.l #1,xxp_strs(a4)    ;slide width = 1
           move.l sought,xxp_tops(a4) ;init value
           move.l #valu,xxp_hook(a4) ;call Valu when slide moves
           jsr _LVOTLSlider(a6)      ;draw slider
           tst.l xxp_errn(a4)
           beq.s Drawn               ;go if ok
           bpl Getval                ;redo if window resized
           TLreqfull                 ;if window too small, make full size
           bra Getval                ;& recycle
          Drawn:
           jsr _LVOTLSlide(a6)       ;draw initial slide posn
           tst.l xxp_errn(a4)
           bne Getval                ;recycle if window resized
          Wait:                      ;wait for user input
           TLwcheck
           bne Getval                ;recycle if window resized
           jsr _LVOTLKeyboard(a6)    ;get response
           cmp.b #$1B,d0             ;done if Esc pressed
           beq Done
           cmp.b #$80                ;reject if not lmb click
           bne Wait
           jsr _TLSlimon             ;if in slider, update slide
           tst.l d0
           bgt Getval                ;recycle if window resized
```

```
          bra Wait                  ;else, wait until Esc pressed
         Done:
          TLreqcls                  ;clear window & return
          rts

         Valu:                      ;serve as xxp_hook when TLSlide called
          move.l xxp_tops(a4),d0
          move.l d0,sought          ;update sought
          move.l a4,a0
          move.l #'    ',(a0)       ;put value in xxp_buff in ascii
          clr.b 4(a0)
          jsr _LVOHexasc(a6)
          move.l #600,d0
          moveq #12,d1
          jsr _LVOTLTRim(a6)        ;show value beside Slider
          rts
```

1.270 tabz

```
                Tabs Toutines

The following routines allow you to draw a set of cards on the

                currently popped window
                , with labelled thumb tabs which the user may
click to bring any of the cards to the front. This allow the user to set
information in categories in a convenient way.

TLTabs          ;render tabs
                TLTabs
                TLTabmon        ;monitor tabs
                TLTabmon
                TLPict          ;draw little picture
                TLPict
                The above routines use rather plain graphic representation, and ↩
                   moreover the
colour scheme is not flexible, but on the other hand it is very easy and
convenient to program.

To use the tabs:

1. First, you must set up: call TLtabs with:
   \1 = the stringnum of the thumbtag labels, each sep by a \
   \2 = the minimum body width
   \3 = the body height (excluding the thumb tags)
   This sets things up, but does not draw the tab cards. You can see the
   total width and total height in xxp_tbbw(a4) and xxp_tbbh(a4)

3. Then, draw the cards, call TLtabs with
   \1 = -1
   \2 = the frontcard (1+)          } After you draw with TLtabs you must
   \3 = xpos                        } then draw teh contents of the body
   \4 = ypos                        } of the cards
   TLtabs will trim the cards to fit the current window size. If the window
   is resized, it TLtabs returns EQ (error).
```

4. Every time you get a TLkeyboard input, if it is $80, then you should call TLtabmon to see if a thumbcard is clicked. If not, TLtabmon returns EQ, with D0=0. Else, TLtabmon returns NE, with D0=1+, being the thumb tag clicked. TLtabmon will also have called TLtabs so the thumb tags will be updated, and the new card (which could be the same as the one which was already at the front) will be blank ready to draw on. (You can also call TLtabs direct at any time to bring a card to the front).

6. When the tabcards are finished with (e.g. the user has clicked a button labelled "Quit" on one of the cards), then you can call TLtab with \1 = 0 to blank the cards.

After you set things up, take care to be consistent with xpos and ypos in you calls to TLtabs or TLtabmon.

Drawing Little Pictures

Tandem.i has a set of 16 little pictures which it uses in its rendering. You can see these in Tandem/Teaching/65.asm. You can draw any of those (e.g. 11 the check mark may be useful) using

```
TLPict
.
```

1.271 tabs

```
()=TLTabs(D0,D1,D2,D3,A4)   [
TLWindow
 must have been called]
```

Sets up/Draws/Removes a set of tab cards on the currently popped window

```
.
```

First, you must call TLtabs to set up the set of tab cards. D0 is a string number of a set of thumb tag titles separated by \'s. e.g.

```
  dc.b 'In\Out\Pending\Too hard',0  ;17
  ....
  moveq #17,d0
```

Place the required body width of the tabcards in D1, and body height in D2. TLTabmon then fills xxp_tbbw(a4), xxp_tbbh(a4) with the total width and height of the tabcards (the width may be more than what you specified in D1, if the thumb-tabs take more than that width to render). TLTabmon also fills xxp_tblw(a4) and xxp_tblh(a4), being the thumbtabs width and heights. The difference between the total width and height and the thumbtab width and height is the body width and height.

n.b. TLTabs uses the font, fontstyle and tspace of the currently popped window for its rendering. So, don't forget to have a consistent font in it each time you call TLTabs/TLTabmon for the same set of tabcards.

Then, draw the tabcards by setting D0=0, and D1 = which tabcard to show (1+), D2 = xpos, D3 = ypos. TLTabs puts the relevant thumbtab at the front, and the body is blank, ready for you to draw its contents.

```
Then, call
                TLtabmon
                 each time you get an input from TLkeyboard with D0=$80
(left mouse click). TLtabmon will return 1+, and call TLtabs with D0=0 and
D1=1+ if a thumbtab was clicked, else returns D0=0.

If the tabcards won't fit on the window they'll be clipped; if the window
is resized, TLtabs returns with an error.

Finally, call TLtabmon with D0=0 and D1=0 to erase the tabcards when they are
finished with.

Call:  To set up:   D0=stringnum  D1=(minimum) body width  D2=body height
       To render:   D0=0  D1=tab(1+)  D2=xpos  D3=ypos
       To clear:    D0=0  D1=0  D2=xpos  D3=ypos

Back:  all regs saved.   xxp_errn(a4)<>0 if window resized.

MACRO: setup:   \1=strnum  \2=min bodywidth  \3=body height
       render:  \1=0  \2=frontcard(1+)  \3=xpos  \4=ypos   (EQ if bad)
       clear:   \1=0  \2=0  \3=xpos  \4=ypos

Example:  See  Tandem/Teaching/65.asm
```

1.272 tabm

```
                D0=TLTabmon(D0,D1,

Monitors the thumbtabs of a set of tabcards set up by
                TLTabs
                ,
which see.

You should call TLtabmon only if TLkeyboard has returned with D0=$80, i.e.
left mouse button clicked.

If TLtabmon finds that one of its thumbtabs has been clicked, it calls TLtabs
to bring that tabcard to the front, and clear the card body ready for you
to draw it, and sets D0=1+ for the tabcard brought to the front. Else,
TLtabmon returns with D0=0.

Call:   A4 as set by Front.i
        D0,D1 are the D1,D2 returned by TLkeyboard
        D2,D3 are the xpos, ypos of the tabcards.

Back:   D0=0 if no thumbtab clicked
        D0=1+ if tabcard 1+ has been clicked (when its body must be drawn)

MACRO:  \1,\2 = TLkeyboard D1,D2
        \3,\4 = xpos,ypos   Sets EQ if no thumbtab clicked.

Example:   [A4 as set by Front.i]    or    [A4 as set by Front.i]
           move.l xxp_tanb(a4),a6          TLkeyboard
           jsr _LVOTLKeyboard(a6)          cmp.b #$80,d0
```

```
           cmp.b #$80,d0                    bne oldtab
           bne oldtab                       TLTabmon d1,d2,#xpos,#ypos
           move.l d1,d0                     beq oldtab
           move.l d2,d1                     subq.w #1,d0
           move.l #xpos,d2                 newtab: ;d0=tab=0+
           move.l #ypos,d3
           jsr _LVOTLTabmon(a6)
           subq.w #1,d0
           bmi oldtab
          newtab: ;d0=tab=0+
```

```
Sample program:    Tandem/Teaching/65.asm
```

1.273 pict

```
                ()=TLPict(D0,D1,D2,A4)             [
                TLWindow
                 must have been called]

Draws on of 16 little icons on the
                currently popped window
                .
(Actually 0-12 are used, 13-15 are for future expansion).
If the icon won't fit on the window, trims to fit.
For samples of each icon, assemble & run Tandem/Teaching/65.asm

Call:     A4 as set by Front.i
          D0 = which (0-11)
          D1 = xpos    } relative to the printable part of the window
          D2 = ypos    }

MACRO:    \1 = which    \2 = xpos  \3 = ypos

Back:     All regs saved.   Sets xxp_errn(a4)=35 if the window was resized.

Example:  [A4 as set by Front.i]      or       [A4 as set by Front.i]
          move.l xxp_tanb(a4),a6               TLpict #6,#10,#5
          moveq #6,d0
          moveq #10,d1
          moveq #5,d2
          jsr _LVOTLPict(a6)

Sample program:    Tandem/Teaching/65.asm
```

1.274 drpd

```
                Drop Down Menu Routine

tandem.library has the following routine for drawing and monitoring a
drop down menu:
```

```
                TLDropdown
                   Draw or monitor a drop down menu

You can specify "cycle" in \8 if you want for a cycle gadget, but this is
only pleasing if you have a few things to choose from. Here are the features
&c of tandem.library dropdown menus:

1. First, you set up a set of strings, being the alternatives.
2. Then, you draw the menu, by calling TLdropdown with \1 = 'draw'.
3. Then, each time you get a TLkeyboard input, if it is $80 (=lmb), then
   call TLdropdown with \1 = 'monitor'. If the user has clicked its down
   arrow, TLdropdown will return NE, and an item from the list will be
   chosen with D0 = 1+. Else, it will return EQ with D0 = 0.
4. When TLdropdown operates, it will return without selection if the
   window is resized. If so, and the user caused it to drop down before
   resizing the window, the drop down will not be erased, so you will have
   to clear the window before redrawing if there is any chance that the
   user resized the window while the menu was dropped down.
5. TLdropdown clips everything to fit if the window is too small. If the
   dropdown doesn't fit on the window, TLdropdown will move it up &/or
   left until it fits, if possible. Thus, you can have drop down menu
   "gadgets" right near the bottom of the window.
6. if the menu drops down, TLdropdown will not return until the user
   does one of the following:
    - presses <Esc>
    - re-clicks the down arrow gadget
    - makes a selection (by clicking a line on the drop down)
    - clicks the close window gadget (if any)
    - resizes the window
7. the drop down has a little slider on it, and the user can use the
   slider to bring items to view if there are too many to fit on the
   drop down.
```

1.275 drop

```
                D0=TLDropdown(D0,D1,D2,D3,D4,D5,D6,D7,A4)
                                                [
                TLWindow
                 must have been called]

Draws or monitors a drop down menu on the
                currently popped window
                .

Read the notes in
                Drop Down Menu Routine
                 before before
continuing.

If you call with D6=0, TLdropdown will make the menu wide enough to
accommodate the widest string. Else, it will make it the value you
suggest, and any too-long strings will be truncated.

If you set D7 < D2, then the drop down will ony be D7 strings, so the user
must use the slider to see all the strings. In any case, the drop down
```

```
cannot be <5 or >14. If D2 is <5, then some lines on the dropdown will
be blank. You can make drop down smaller or larger for design/aesthetic
considerations.

If you set D7=-1, then for draw there will be a cycle icon on the gadget,
and if you monitor it it will not drop down a menu but cycle through the
choices. Else, there will be a down arrow icon, and a menu will drop down.
You should ony use cycle if there are very few choices (if then).

TLdropdown is not actually a requester, since it does not use its own window.
It copies the region under the dropdown to xxp_EBmap, and on exit copies
it back again. Caution: If the user resizes the window while the dropdown
is down (an odd thing to do!) then TLdropdown does not erase the dropdown
before it returns. Thus, you should then clear the window before re-drawing
it.

Call:  A4 as set by Front.i
       D0 = 0 if draw   -1 if monitor
       D1 = 1st string num
       D2 = number of strings (= number of choices)
       D3 = operative choice (1+)
       D4 = xpos
       D5 = ypos
       D6 = max characters (0 for automatic - see above)
       D7 = maximum drop (5 to 14), or -1 if cycle only (see above)

Back:  if D0 was 0, D0=0
       else, D0=0 if down arrow gadget was not clicked
             D0=1+ if item 1+ selected (if user cancelled, = calling d3)

MACRO: \1 = 'draw' or 'monitor', default 'monitor'
       \2,\3 = 1st string num, no. of strings
       \4 = operative choice (1+) default #1 (must be <= \3)
       \5,\6 = xpos,ypos
       \7 = max chrs, or 0 for automatic, default 0
       \8 = max dropdown (5 to 14), or 'cycle', default #7
       (saves all if draw, else all except result in D0)
```

1.276 cust

```
                Making a Custom Requester

TLReqredi       ;prepare for requester
                TLReqredi
                TLReqchek       ;requester dims
                TLReqchek
                TLReqon         ;render requester
                TLReqon
                TLReqoff        ;requester off
                TLReqoff
                tandem.library has three subroutines to help you to make custom ↩
                   requesters.
Here is the skeleton of a requester:
```

```
 sub.l #24,a7              ;create dummy part xxp_wsuw (it_SIZEOF+4)
 move.l a7,a5              ;a5 points to dummy xxp_wsuw
 TLreqredi                 ;set pop window, initialise things
 beq .bad                  ;(go if init fails - unlikely)

 ......                    ;[calculate requester dimensions]

 TLreqchek                 ;check requester size &c
 beq .bad                  ;go if won't fit
 tst.w xxp_ReqNull(a4)     ;go if null
 beq .null
 TLreqon                   ;turn requester window on
 beq .bad                  ;go if can't

 .....                     ;[do the requester's thing]

.good:                     ;here to close down ok
 TLreqoff                  ;close requester window, pop old window if any
.null:
 moveq #-1,d0              ;signal ok
 bra.s .done
.bad:
 moveq #0,d0               ;signal bad
.done:
 move.w #-1,xxp_reqNull(a4)
 TLwslof                   ;clear all message buffers
 add.l #24,a7              ;remove dummy IntuiText
 movem.l (a7)+,d1-d7/a0-a6 ;D0=0 if bad, else 1+=choice
 rts
```

The routines are:

TLReqredi
initialises things

TLReqchek
set requester dimensions

TLReqon
turn the requester on

TLReqoff
turn the requester off

The reason for the "dummy xxp_wsuw" above, is to allow your requester to open direct on the screen, if that is what you want. If there is a

currently popped window
the requester will attach to it, else it will attach direct to the screen. Note that before you can call any of the above you must have called

TLWindow
at least once (e.g. by TLwindow #-1).

For a more complete discussion, see also the notes at the beginning of the

```
sample program   Tandem/Teaching/69.asm
```

1.277 qrdi

```
                D0=TLReqredi(A4,A5)                [@"@TLWindow" link wind] must  ↩
                   have been called]

Sets things up for creating a custom requester.

See also:
                Custom Requesters

                Primitive Routines
                Call:   A4 as set by Front.i
        A5 a 24 byte workspace (usually in stack)

Back:   D0=-1 if set up ok
        D0= 0 if bad (unlikely)

MACRO:  \1 = 24 byte workspace        sets EQ if bad

Example:      see Tandem/Teaching/69.asm
```

1.278 qchk

```
                D0=TLReqchek(D2,D3,A4)             [
                TLReqredi
                 must have been called]

Checks the dimensions & position of a requester

See also:
                Custom Requesters

                Primitive Routines
                Call:   A4 as set by Front.i
        D2 proposed width
        D3 proposed height

Back:   D0 = -1 if fits ok
        D0 =  0 if too large for screen

MACRO:  \1 = proposed width,  \2=proposed height   set EQ if bad

Example:     see Tandem/Teaching/69.asm
```

1.279 qonn

```
                D0=TLReqon(A4)                      [
                TLReqchek
                 must have been called]

Creates a requester window, puts a border around it

See also:
                Custom Requesters

                Primitive Routines
                Call:   A4 as set by Front.i
        A5 = 24 byte workspace as set by TLReqredi

Back:   D0 = -1 if successful
        D0 =  0 if out of mem

MACRO:  \1 = 24 byte workspace as set by TLReqredi    sets EQ if bad

Example:     see Tandem/Teaching/69.asm
```

1.280 qoff

```
                D0=TLReqoff(A4)                     [
                TLReqon
                 must have been called]

Closes a requester window. OK to call if none open

See also:
                Custom Requesters

                Primitive Routines
                Call:   A4 as set by Front.i

Back:   -

MACRO:  no parameters   no return

Example:     see Tandem/Teaching/69.asm
```