Documentation LDS

Syntaxe

Manuel du compilateur de fichier L.D.S
(Langage de Description Symbolique).

Tableau de mise à jour

Version	Date	Nom	Objet
01	29.04.2003	M. Julienne	Création du document

1 Préface

1.1 Documents

1.1.1 Documents de références

Acronyme	Titre / Désignation	Référence	version	Origine
DR1	recommandations Z100 à Z104 de 1984		1	CCITT

1.1.2 Documents applicables

Acronyme	Titre / Désignation	Référence	version	Origine
DA1				

2 Présentation générale

2.1 Domaine d'application

Ce document décrit la syntaxe et le mode d'utilisation des compilateurs LDS. Les produits concernés par cette documentation sont les progiciels suivants :

- **LDS_C**
  Compilateur générateur de code source au format C ansi.

- **LDSGRAPH**
  Compilateur générateur d'un code spécifique définissant un tracé graphique des spécifications.

- **HPPLOT**
  Gestionnaire de table traçante HP 7475 ou Postcript permettant le tracé du code généré par LDSGRAPH.

2.2 Rôle des outils LDS

Les outils LDS s'appuient sur un sous ensemble du "Langage De Spécification et de description fonctionnelles (LDS)" défini par les recommandations Z100 à Z104 du CCITT (Comité Consultatif International et Téléphonique) en 1984.

Ce sous ensemble représente ce dont on a besoin pour travailler dans de bonnes conditions sur un microprocesseur 8 bits (compte tenu de la nature et de la complexité relative des projets sur ce type de microprocesseur) et que l'on peut raisonnablement implanter sur un séquenceur temps réel tournant sur 8051.

Par la suite, on désignera ici sous le terme de LDS l'interprétation qui en est faite dans nos outils et non la lettre de la spécification telle qu'elle est décrite dans les recommandations CCITT.

En imposant une formalisation textuelle des spécifications fonctionnelles des différentes tâches de l'application à réaliser, les outils LDS permettent d'automatiser certains travaux liés à ces spécifications :

- La visualisation sous forme graphique (outils LDSGRAPH et HPPLOT).
- La validation au moyen d'un simulateur LDS non encore réalisé aujourd'hui.
- La génération de code source (outils LDSC).
- Les outils LDS interviennent dans toutes les phases du projet.

2.2.1 Phase de spécifications

Ils permettent la mise en forme des spécifications fonctionnelles de manière standardisée et génèrent une sortie graphique de très bonne qualité.

2.2.2 Phase de réalisation

Ils assurent la génération du code source du squelette de l'application y compris celle des tables d'automates.

2.2.3 Phase de maintenance

Le code généré, s'il ne contient "que" le squelette de l'application fourni néanmoins des fichiers "autonomes" sur lesquels il n'est pas nécessaire d'intervenir manuellement.

Toute modification de spécification se trouve donc automatiquement répercutée aux niveaux source et documentation par simple compilation.

S'ils permettent la génération de code source, Les compilateurs LDS restent néanmoins des outils de spécification et non des langages de programmation. Ils génèrent le "squelette" de d'application, C'est à dire le séquencement des tests et des appels de procédures et se reposent sur les langages de programmation "cibles" (assembleur ou C) pour les définitions de type, le codage des tests et procédures et l'édition des liens.

Cette architecture permet de bénéficier de l'ensemble des qualités respectives des langages de programmation utilisés (contrôle du facteur d'expansion et de la vitesse d'exécution pour l'assembleur et programmation en langage de haut niveau pour le C).

Les compilateurs LDS n'assurent en particulier aucun traitement direct sur les données, ce qui permet d'éviter une double définition (au niveau LDS et au niveau langage de programmation). Par contre, il est transparent par rapport aux instructions d'affectation et de test (pour le C uniquement) et permet le passage de paramètres à l'appel des tests et procédures (pour toutes les versions).

2.3 LDS et la programmation par automates

Par nature, le LDS fait amplement appel au concept d'automate.

Un automate est une tâche (ou processus) pouvant prendre un certain nombre d'états qui soumis à un événement effectue un traitement qui dépend de l'état dans lequel elle est.

Le concept d'automate est très bien adapté à la description fonctionnelle puisqu'il oblige à faire la liste des événements auxquels sont soumis les différentes tâches et à définir un traitement pour chaque couple état/événement.

Il est également précieux pour la mise au point, les tests modulaires et les tests de validation, puisque pour tester une tâche, il suffit en principe de tester toutes les "cases" de la matrice d'automate (tous les couples états/événements).

Toute tache codée en LDS sera considérée comme un automate.

Le cas particulier des taches ne possédant qu'un seul état permet de donner une apparence d'automate à des tâches "classiques" possédant un ou plusieurs événements correspondant à des points de réveil différents .

Le LDS permet de définir , outre les couples états /événements et les traitement liés , des procédure qui peuvent être appelées soient dans les traitements eux même , soit par des procédures « externe » telles que les handlers ou le séquenceur.

On peut donc en ce sens, utiliser également les outils LDS comme simple "générateurs d'organigrammes"... sachant générer **du code source !**

3 Le fichier source LDS

3.1 Le jeu de caractères

3.1.1 Format général d'une instruction LDS

La séquence de caractères délimitant les instructions est la séquence CR/LF.

Une instruction LDS tient donc obligatoirement sur une ligne (de longueur maximum de 132 caractères) et il y a donc également une seule instruction LDS par ligne.

3.1.1.1 Les séparateurs

Les caractères suivants sont considérés comme séparateurs par les compilateurs LDS :

( ) , % *

Ainsi que l'espace et la tabulation.
Les champs d'une instruction LDS sont séparés par un nombre quelconque (non nul) de séparateurs.

3.1.1.2 Les commentaires

Des commentaires peuvent être insérés à tout endroit du fichier source. Ils doivent être encadrés par la séquence de début de commentaire :
"/*" et bar celle de fin : "*/".
La fin de ligne tient également lieu de séquence de fin de commentaire.
Les lignes vides ou ne contenant que des commentaires sont ignorées par les compilateurs.

/* lignes de commentaires */

```
/*
/* Fichier : Physique.sdl
/*
/*
```

3.1.1.3 Caractère d'échappement

Le caractère d'échappement permet d'insérer des séparateurs dans un champ.

Il est représenté par le caractère "|".

Après avoir rencontré ce caractère, l'analyseur syntaxique des compilateurs considère comme faisant partie du champs en cours d'analyse tous les caractères (séparateurs compris) jusqu'au prochain caractère d'échappement.
Ces caractères seront filtrés et ne seront pas présents dans le code source généré.

```
task |i=i+1|
output console |'Version 1.2'|
```

3.1.1.4 Les Champs

Les champs d'une instruction LDS sont donc constitués de tous les autres caractères. Il n'y a, au niveau LDS aucune autre restriction sur les identificateurs.
Cependant, les identificateurs générés au niveau du code source reprenant ceux du fichier LDS, en pratique, on appliquera au niveau LDS les restrictions applicables au langage de programmation utilisé.

Les différents "champs" d'une ligne LDS sont :

- un mot clé (obligatoire, quel que soit l'instruction),
- un opérateur,
- des opérandes

La longueur d'un champ est limitée à 32 caractères et le nombre de champs par instruction à 16.

```
If cartePresente(numero_de_carte)
then
     task lectureCaractere
     task |i=3|
fi
```

3.1.1.5 Les mots clés

Les mots clés sont les suivants :

ALTERNATIVE, CASE, ELSE, ENDALTERNATIYE, ESAC, EXIT, FI, IF, INPUT, NCALL, NEND, NPROC, OF, OTHERWISE, OUTPUT, PROCEND, PROCESS, REPEAT, STATE, STATEND, TASK, THEN, UNTIL, WHILE, WHILEND, XCALL, XEND, XPROC START.

Ils ne sont analysés en temps que tels qu'en premier champ d'une ligne LDS et ne constituent donc pas des mots réservés en ce sens qu'ils peuvent être utilisés comme identificateurs.

3.1.1.6 Opérateurs et opérandes

Le deuxième champ, lorsqu'il est présent et que le contexte (c'est à dire le mot clé) le justifie est considéré comme l'opérateur.
Lors d'un appel de procédure, c'est lui qui est interprété comme étant le nom de la procédure.
Les autres champs sont considérés comme opérandes.
Ils désigneront des identificateurs (noms de tâche, d'état, d'événement...) ou des paramètres (appel de procédure ou test).

3.2 Les blocs d'une description LDS

3.2.1 Structure générale

La description LDS est une description par bloc, c'est à dire que la description de chaque objet est à chaque niveau (tâche, état, traitement ou procédure, test, boucle), limité par une instruction de début et une instruction de fin.

3.2.1.1 Bloc " tâche " (PROCESS / PROCEND'')

La description d'une tâche LDS doit être contenue dans un fichier unique.

Elle est encadrée par Les instructions :

PROCESS <nom de la tâche >
PROCEND <nom de la tâche>

Ce bloc ne peut être inclus dans aucun autre. Les blocs "état" et "procédure" sont seuls directement inclus dans le bloc « tache ».

Process CONTROLE
.../...
Procend CONTRÔLE

3.2.1.2 Bloc " état " (‘‘ STATE / STATEND ")

La description d'un état est encadrée par les instructions :

STATE <non de 1 ' état>
STATEND <nom de 1 ' état>

Il est directement inclus dans le bloc tâche. Les blocs "événement" sont seuls directement inclus dans le bloc "état".

STATE VEILLE
.../...
STATEND VEILLE

3.2.1.3 Bloc « événement » (« INPUT / EXIT »)

La description du traitement associé à un état et un événement se fait à l'intérieur du bloc définissant l'état par un bloc encadré par les instructions suivantes :

INPUT <nom de l'événement> [, [nom de l'événement]...]
EXIT <nom du nouvel état>
ou
XCALL <nom de procédure> (voir appel de procédure)

Un seul bloc "événement" peut être utilisé en cas de traitement commun à plusieurs événements.
Tous les blocs de tests' de boucles' les appels de procédures et les envois d'événement peuvent être inclus dans ce bloc.

INPUT evt1, evt2
EXIT NouvelEtat
INPUT evt3
.../...
EXIT MemeEtat

3.2.1.4 Blocs " procédure " (‘‘ X PROC / X E N D’’ e t " NPROC / NEND")

On distingue deux types de procédures suivant que l'on défini ou non un nouvel état de sortie à l'intérieur de la procédure.
Une procédure dans laquelle l'état de sortie de l'automate n'est pas défini est encadrée par les instructions :

NPROC <nom de la procédure>
NEND <nom de la procédure>

Une procédure dans laquelle l'état de sortie est défini est encadrée par les instructions :

XPR0C <nom de la procédure>
XEND <nom de la procédure>

Tous les blocs de tests, de boucles, les appels de procédures et les envois d'événement peuvent être inclus dans ces blocs.

```
NPROC Procedure 1
        . . . / . . .
NEND Procedure 1

XPROC Procedure 2
        . . ./. . .
        EXIT NouvelEtat
XEND Procedure 2
```

3.2.2 Appel de Procédure

On distinguera trois types d'appel de procédure

- Non défini au niveau LDS.
- Ne définissant pas un nouvel état.
- Définissant un nouvel état.

3.2.2.1 Procédure non définie au niveau LDS

De telles procédures (définies directement au niveau programmation) sont appelées par l'instruction :

TASK <nom de la procédure> [[paramètre],...]

TASK EnvoiCaractere('I', PortSerie)
TASK | i=i+1|
TASK (i=i+1)

3.2.2.2 Procédure LDS ne définissant pas un nouvel état

Ces procédures sont appelées par l'instruction :

NCALL <nom de la procédure>

NCALL Procedure
NCALL EnvoiCaractere

3.2.2.3 Procédure LDS définissant un nouvel état

Ces procédures sont appelées par l'instruction :

XCALL <nom de la procédure>

XCALL TraitementCaractere
XCALL TestEtRecoitCaractere

3.2.3 Les événements

L'envoi d'un événement à une autre tâche est spécifié par l'instruction :

OUTPUT [[paramètre]..]

La signification des paramètres d'envoi est laissée à l'initiative de l'utilisateur.

Exemples d'envoi d'événement ("OUTPUT")

OUTPUT event (Tache, Evenement, Parametre)
OUTPUT timer (Tache1, Evenement3, Parametre5, Durée, SECONDE)
OUTPUT timer(Tache1, Evenement3, Parametre5, Durée, MS)

3.2.4 Les tests

3.2.4.1 Le test binaire (bloc "IF/FI")

Un test donnant lieu à deux alternatives (OUI/NON) est codé par le bloc suivant :

IF <nom du test>, [[paramètres]...]
THEN
Instructions exécutées si le test est positif
ELSE
Instructions exécutées si le test est négatif
FI

L'opérateur associé à l'instruction IF permet au compilateur d'identifier le test à appeler.
Ce peut être un appel de procédure extérieure à LDS (écrite en langage de programmation), un appel de macro instruction (dans le cas de l'assembleur) ou un test entre deux expressions (dans le cas du C).

La branche **ELSE** est facultative.

```
IF TestCaracteres(car, 'A')
THEN
ELSE
        . . . / . . .
FI
IF | i=5 |
THEN
        . . . / . . .
FI
IF ( j=5)
THEN
        . . . / . . .
FI
```

3.2.4.2 Le test à alternatives multiples (bloc "CASE/ESAC")

Un test donnant lieu au moins deux alternatives est codé par le bloc suivant :

CASE <nom du test>, [[paramètres]...]
OF [[valeurl....].

Instructions exécutées si la valeur à tester figure dans la liste définie par l'instruction OF

[lOF ttvaleur]...]

Instructions exécutées si la valeur à tester figure dans la liste définie par l'instruction OF

]...][OTHERWISE

Instructions exécutées si la valeur à tester n'a été trouvée dans aucune des listes définies dans les instructions
OF |ESAC

Les champs nom du test et paramètres ont la même signification que pour le test IF

```
case EtatRX
of debut
        if CaractereDebut
        then
                task NouvelEtatRx(longueur)
        fi
of longueur
        task MemoLongueurRX
        task InitRX
        task NouvelEtatRX(message)
esac

case X
 of 5
        task Traite5
 esac
```

Boucles

3.2.4.3 Boucle avec test en début de boucle (bloc "WHILE/WHILEND")

Ces boucles sont codées par le bloc :

WHILE <nom du test>, [[paramètres]...]
Instructions exécutées dans le corps de la boucle
WHILEND

Les champs nom du .test et paramètres ont la même signification que pour le test IF.

```
task initCompteur(5)
while compteurNonNul
        task decrementeCompteur
        task traite
whilend
task | Compteur=5 |
while |compteur>0 |
        task compteur=compteur-1|
        task traite
whilend
```

3.2.4.4 La boucle avec test en fin de boucle (bloc "REPEAT/UNTIL")

Ces boucles sont codées par le bloc :

REPEAT
Instructions exécutées dans le corps de la boucle
UNTIL <nom du test>, [[paramètres]...]

Les champs nom du test et paramètres ont la même signification que pour le test IF.

```
task initCompteur(5)

repeat
        task decrementeCompteur
        task traite
until compteurNul
task (Compteur=5 )
repeat
        task |compteur=compteur-1|
         task traite
until |compteur=0|
```

3.2.5 Compilation conditionnelle (bloc « ALTERNATIVE / ENDALTERNATIVE »)

Le codage d'une structure de compilation conditionnelle (qui sera évaluée au moment de la compilation du source généré par le compilation LDS) est réalisée par le bloc suivant :

ALTERNATIVE <option>
OF t[option]...] .
Instructions compilées si l'option de test est trouvée dans la liste

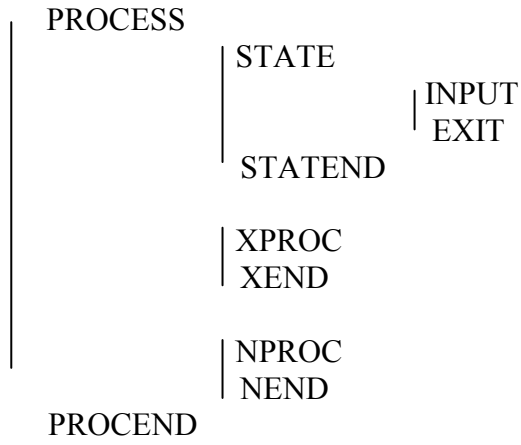
[[OF [[option]...]
Instructions compilées si 1' option de test est trouvée dans la liste
, . . .] [OTHERWISE
Instructions compilées sinon
]ENDALTERNATIVE

Les options testées sont des options du langage de macro (dans le cas de l'assembleur) ou des composantes de noms de "switch" (dans le cas du C).
Le source générée produira donc une compilation conditionnelle qui sera évaluée par l'assembleur ou le compilateur C.

```
alternative typeEquipement
of ETTD task
        Traitement
otherwise
        task Traitement
endAlternative
```

3.3 Imbrication des blocs

Le schéma d'imbrication des blocs de plus haut niveau est le suivant :



A l'intérieur de ces blocs, on peut trouver une imbrication (d'une profondeur maximum de 50 niveaux) de blocs tests ou bouleens et d'instructions simples.

Ces imbrications sont réglementées par les restrictions classiques de non chevauchement de blocs (c'est à dire qu'un bloc doit se terminer à l'intérieur du bloc dans lequel il a débuté).

La seule exception provient de la possibilité de définir un nouvel état (instructions "EXIT" ou "XCALL") à l'intérieur des branches d'une structure de test.
Cette possibilité permet de définir plusieurs états de sortie pour un seul couple état/événement suivant des tests décrits au niveau LDS.

Par soucis de cohérence, si un état de sortie est défini dans une branche d'un test, toutes les autres branches doivent également comporter une- instruction "EXIT" ou "XCALL". De plus, pour chacune des branches, les instructions "EXIT" ou "XCALL" marquent la fin de la description de la branche et l'instruction qui suit doit obligatoirement être une instruction définissant une autre alternative.
Dans ce cas, les branches "ELSE" et "OTHERWISE" sont rendues obligatoires.

Un bloc "INPUT/EXIT" ainsi structure est considéré comme terminé lorsque le dernier bloc de test est clos.

Exemple de fichier LDS complet

```
#include "c:\v1_boot\t\inc\usine.inc"
#define process Usine
nproc TraiteErreur
      case (Param)

      of ERREUR_ZONE_NON_EFFACE
            task printf("\nError: flash area not erased\n>")

      of ERREUR_TEMPO_EFFACEMENT_MAX
            task printf("\nError: erased time_out\n>")
```

```
      of ERREUR_DEFAUT_ZONE_A_EFFACER
            task printf("\nError:  flash area not erased (1)\n>")

      of ERREUR_DEFAUT_PROGRAMMATION
            task printf("\nError: programming default\n>")

      otherwise
            task printf("\nError: not defined : %d\n>",Param)
      esac

nend TraiteErreur


xproc TraitementCommande
      case AnalyseArgumentEtValidite
            of  CDE_INCONNU CDE_ERREUR
                  task AfficheLigneCommande
                  task printf("\nError: INVALID COMMAND\n>")
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit -
            of MODE_NON_AUTORISE
                  task printf("ACCESS DENIED\n>")
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit -
            of CDE_TRACE_ON
                  task (TableauTrace[TableauArgument[2].ValArg] = OUI)
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit -
            of CDE_TRACE_OFF
                  task (TableauTrace[TableauArgument[2].ValArg] = NON)
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit -
            of CDE_ETAT_FLASH
                  output event(Memoire, DebugAfficheEtatFlash,0)
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit -

            of CDE_SET_SERIAL_LINK_TERM
                  if (TraiteSetSerialLinkTerm() == NOK)
                  then
                        task printf(StrBadArgument)
                  fi
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit -

            of CDE_D_LOAD_SOFT_RELEASE
                  task (VarUsine = ZONE_CPU)
                  task (AutomateHandlerUsine = INIT_RECEPTION_FICHIER)
                  task printf(StrAttenteDuFichier)
                  output timer(Usine,TempoAttenteFichier,0,5,SECONDE)
              exit ReceptionFichier

            of CDE_D_LOAD_SOFT_BOOT
                  task (VarUsine = ZONE_BOOT)
                  task (AutomateHandlerUsine = INIT_RECEPTION_FICHIER)
                  task printf(StrAttenteDuFichier)
                  output timer(Usine,TempoAttenteFichier,0,5,SECONDE)
              exit ReceptionFichier

            of CDE_D_LOAD_FPGA_RELEASE_FPGA_LUT
```

```
                  task (VarUsine = ZONE_LUT)
                  task (AutomateHandlerUsine = INIT_RECEPTION_FICHIER)
                  task printf(StrAttenteDuFichier)
                  output timer(Usine,TempoAttenteFichier,0,5,SECONDE)
            exit ReceptionFichier
          of CDE_D_LOAD_FPGA_RELEASE_ALE_16TAP
                  task (VarUsine = ZONE_ALE_16TAP)
                  task (AutomateHandlerUsine = INIT_RECEPTION_FICHIER)
                  task printf(StrAttenteDuFichier)
                  output timer(Usine,TempoAttenteFichier,0,5,SECONDE)
            exit ReceptionFichier
          of CDE_D_LOAD_FPGA_RELEASE_ACUSCALE
                  task (VarUsine = ZONE_ALE_ACU_SCALE)
                  task (AutomateHandlerUsine = INIT_RECEPTION_FICHIER)
                  task printf(StrAttenteDuFichier)
                  output timer(Usine,TempoAttenteFichier,0,5,SECONDE)
            exit ReceptionFichier


          of CDE_GET_MEMORY
                  if (DumpMemoire() !=OK)
                  then
                        task printf(StrBadArgument)
                  fi
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit -

          of CDE_SET_MEMORY
                  task SetMemoire
                  task printf (StrChaineRetour)
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit -

          otherwise
                  task printf("COMMAND ERROR\n>")
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit -
      esac
xend TraitementCommande

start Usine
      task (TableauTrace[PROCESS] = OUI)
      task InitUsine
      task (DspFpgaCtrlH.Bit.Led =LED_ORANGE)
      task (ModeCommande = MASK_MODE_REGLAGE)
      task (AutomateHandlerUsine  = ATTENTE_COMMANDE)
      exit AttenteMemoire

state AttenteMemoire
      input ApplicatifChargeOK
            if (PresenceCavalier() == NON)
            then
                  task JeSaute()
                  exit -
            else
                  task AfficheVersion
                  output event(Memoire,JeNeSautePas,0)
                  exit Operationnel
            fi
      input ApplicatifNOK
```

```
            task AfficheVersion
            exit Operationnel
      input -
            task TraitementEvenementNonAttendu
            exit -


statend AttenteMemoire

state Operationnel

      input MessageErreur
            ncall TraiteErreur
            exit -
      input ErreurDebordement
            task (AutomateHandlerUsine = ATTENTE_COMMANDE)
            exit -

      input TrameCorrecte
            case AnalyseTrame
                  of ARGUMENT_TROP_LONG
                        task printf("ARGUMENT TOO LONG\n>")
                        task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                        exit -
            of CDE_VIDE
                        task printf(">")
                        task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                        exit -
                  otherwise
                        /* PLUS_DARGUMENT */
                        xcall TraitementCommande
            esac

      input -
            task TraitementEvenementNonAttendu
            exit -
statend Operationnel


state ReceptionFichier
      input MessageErreur
            ncall TraiteErreur
            exit -
      input TempoAttenteFichier
            task StopTimer (TempoAttenteByte)
            task (AutomateHandlerUsine = ATTENTE_COMMANDE)
            task printf("TIME_OUT DOWNLOAD FILE\n>")
            exit Operationnel

      input TempoAttenteByte
            task (AutomateHandlerUsine = INIT_VERIFICATION_FICHIER)
            task printf(StrAttenteDuFichier)
            output timer(Usine,TempoAttenteFichier,0,5,SECONDE)
            exit VerificationFichier

      input ErreurFichierTropGros
            task StopTimer (TempoAttenteByte)
            task printf("ERROR SIZE FILE\n>")
            task (AutomateHandlerUsine = ATTENTE_COMMANDE)
            exit Operationnel
```

```
      input -
            task TraitementEvenementNonAttendu
            exit -

statend ReceptionFichier

state VerificationFichier
      input MessageErreur
            ncall TraiteErreur
            exit -
      input TempoAttenteFichier
            task StopTimer (TempoAttenteByte)
            task (AutomateHandlerUsine = ATTENTE_COMMANDE)
            task printf("TIME_OUT DOWNLOAD FILE\n>")
            exit Operationnel

      input TempoAttenteByte
            if (TestVerificationFichier() == OK)
            then
                  task (AutomateHandlerUsine = ATTENTE_USINE)
                  task printf("FILE PROGRAMMING\n>")
                  output event(Memoire,SauvegardeFichier,VarUsine)
                  exit AttenteFinProgrammation
            else
                  task printf("DOWNLOAD FILE ERROR\n>")
                  task (AutomateHandlerUsine = ATTENTE_COMMANDE)
                  exit Operationnel
            fi
      input ErreurFichierTropGros
            task StopTimer (TempoAttenteByte)
            task printf("ERROR SIZE FILE\n>")
            task (AutomateHandlerUsine = ATTENTE_COMMANDE)
            exit Operationnel

      input -
            task TraitementEvenementNonAttendu
            exit -

statend VerificationFichier

state AttenteFinProgrammation
      input MessageErreur
            ncall TraiteErreur
            exit -
      input ProgrammationFichierOK
            task printf("FILE PROGRAMMING OK\n>")
            task (AutomateHandlerUsine = ATTENTE_COMMANDE)
            exit Operationnel

      input TimeOutProgrammationFichier
            task printf("ERROR: TIME OUT FILE PROGRAMMING %d\n>",Param)
            task (AutomateHandlerUsine = ATTENTE_COMMANDE)
            exit Operationnel

input ErreurProgrammationFichier
            task printf("ERROR: FILE PROGRAMMING\n>")
            task (AutomateHandlerUsine = ATTENTE_COMMANDE)
            exit Operationnel
```

```
/*    input TempoAttenteByte
/*          exit -

      input -
            task TraitementEvenementNonAttendu
            exit -

statend AttenteFinProgrammation



procend Usine
```

4 LDSGRAPH

4.1 Appel

L'appel du compilateur LDSGRAPH se fait par la commande :

LDSGRAPH <fichier LDS> <fichier graphique>

LDSGRAPH aiguille.sdl aiguille.gra
LDSGRAPH d ;\ca\routage.sdl d :\ca\routage.gra

4.2 Spécificité

Il n'y a aucune restriction en ce qui concerne le jeu d'instructions ou la syntaxe des champs.
Par contre l'édition graphique se faisant au moyen de blocs élémentaires (ayant un graphisme très proche de celui défini par le CCITT) de taille constante' il est physiquement impossible de "caser" dans ces blocs une chaîne de caractères trop longue.
Pour cette raison le texte édité dans ces blocs est tronqué à 39 caractères (3 lignes de 13 caractères).

Ce texte représente tous les champs opérateur et opérandes de l'instruction et est édité tel qu'il figure dans le fichier source (avec ces séparateurs et ses distinctions entre minuscules et majuscules).

4.3 Directives

La seule directive reconnue est la suivante :

XFORMAT [[validité format],...l

Elle permet de définir quels sont les formats de papier autorisés lors du tracé. 4 formats sont traités :

- A4 : correspond à une feuille A4 prise dans le sens habituel (portrait)

- A4R : feuille A4 prise dans le sens horizontal (paysage)

- A3 : feuille A3 prise dans le sens vertical

- A3R : feuille A3 prise dans le sens horizontal

Table 3.2: Exemples de directives "%FORMAT"

%FORMAT A4, -A3 t A4R, -A3R valide les formats A4 et A4R et invalide les formats A3 et A3R

%FORMAT A3, -A4 valide le format A3, invalide le format A4, les autres formats restent validés ou non comme ils l'étaient avant la directive.
Par défaut, A4R est seul valide.
Lorsque plusieurs formats sont validés, le compilateur choisi page par page celui qui utilise le moins de feuilles. Lorsque plusieurs formats utilisent le même nombre de feuilles, l'ordre de préférence est le suivant : A4, A4R, A3 puis A3R.

5 HPPLOT

5.1 Appel

L'appel de HPPLOT se fait par la commande :

HPPLOT <fichier graphique>
<fichier graphique> fichier généré par LDSGRAPH

HPPLOT aiguille.gra

HPPLOT d:\ca\routage.gra

5.2 Utilisation

HPPLOT génère un fichier poscript (y compris le choix du format de papier) en fonction des directives de compilation utilisées pas LDSGRAPH.

6 LDS_C

Le programme écrit en langage C fonctionne sur un PC dans un environnement sous DOS (fonctionne sur WIN95 et NT).

Syntaxe du compilateur

LDS_C.exe –i[fichierSrc] –d[FichierDebug] –o1[FichierDest1] –o2[FichierDest2] –r[FichierRap]

- ***FichierSrc*** correspond au nom du fichier source écrit en L.D.S (paramètre obligatoire).
- ***FichierDebug*** correspond au nom du fichier de débugage généré lors de la compilation (paramètre optionnel).
- ***FichierDest1*** correspond au nom du fichier d'automate généré lors de la compilation. (paramètre obligatoire).
- ***FichierDest2*** correspond au nom du fichier généré lors de la compilation. (paramètre obligatoire).
- ***FichierRap*** correspond au nom du fichier de rapport généré lors de la compilation. (paramètre optionnel).

LDS_C c:\source\aiguille.sdl c:\cible\aiguille.C1 c:\cible\aiguille.C2

Comment assembler le fichier source généré

Le fichier source généré ne contient aucune déclaration concernant les macro-instructions sous-programmes ou paramètres utilisés.
Il doit donc être utilisé comme "include" dans un fichier principal contenant les définitions des sous programmes appelés soit par déclarations en référence externe soit par définition directe (écriture).
Ce fichier principal devra également définir toutes les variables utilisées et si besoin, déclarer en "public" les données nécessaires au séquenceur.

Si le fichier de déclaration contient des définitions de macro-instructions il pourra également être utilisé en "include" dans ce fichier principal.

La structure de ce fichier pourra donc être la suivante :

FICHIER PRINCIPAL
Déclarations
Définition des procédures utilisées
#include(<***FichierDest1*** >)

FICHIER DE DECLARATIONS
Déclarations
Définition des macro-instructions
#include(<fichier source généré>)

FICHIER SOURCE GENERE
Squelette de l'application
Définition (équivalences) de tâche, états, événements et nombre d'états
Table d'automate.

Exemple :

```
/* ---------------------------------------------------------------*/
/* Fichier: surveilSignal.c                                       */
/*                                                                */
/* ---------------------------------------------------------------*/
 #include "surv_Sig.C2"
void Fonction1(void)
{….}

void FonctionN(void)
{….}

#include "surv_Sig.C1"
```

Points d'entrée

Un point d'entrée est généré pour chaque bloc « START /EXIT» « INPUT/EXIT », « NPROC/NEND » ou « XPROC/XEND ».

Dans le cas des blocs « NPROC/NEND » et « XPROC/XEND », le nom du point d'entrée est celui de la procédure (défini dans l'instruction « NPROC » ou « XPROC »).

Dans le cas du bloc « INPUT/EXIT », le nom du point d'entrée est formé comme suit :

P_<numéro de tâche>_<numéro d'état>_<numéro d'événement>

Ces trois numéros étant codés en décimal, sans 0 à gauche des chiffres.
Chacun de ces blocs est en outre terminé par un point de sortie unique générant une instruction "RET" (ou "RETI" dans le cas d'un bloc "NPROC/NEND" compilé avec la directive "XINTERRUPT").

Dans le cas des blocs "INPUT/EXIT" ou "XPROC/XEND", ce point de sortie est généré soit par l'instruction "EXIT" si elle n'est pas incluse dans une branche d'une structure de test, soit par la fin de cette structure de test sinon.
Dans le cas des blocs "NPROC/NEND", ce point de sortie est généré par l'instruction "NEND».