Écrire un simulateur pour le système SIMH
Révisé le 22 septembre 2008 pour SIMH V3.8-1

AVIS DE DROIT D'AUTEUR

L'avis de copyright suivant s'applique au code source, au code binaire et à la documentation du SIMH :

1. Vue d'ensemble

Le SIMH (simulateurs d'histoire) est un ensemble de programmes portables, écrits en C, qui simulent divers ordinateurs intéressants d'un point de vue historique. Ce document décrit comment concevoir, écrire et vérifier un nouveau simulateur pour le SIMH. Il ne s'agit pas d'une introduction à la philosophie ou au fonctionnement externe du SIMH, et le lecteur doit être familiarisé avec ces deux sujets avant de poursuivre. Il ne s'agit pas non plus d'un guide sur la conception ou le fonctionnement interne du SIMH, sauf dans la mesure où ces domaines interagissent avec la conception du simulateur. Ce manuel présente et explique plutôt la forme, la signification et le fonctionnement des interfaces entre les simulateurs et le logiciel de contrôle des simulateurs du SIMH. Il propose également quelques suggestions pour l'utilisation des services offerts par le SIMH et explique les contraintes auxquelles sont soumis tous les simulateurs fonctionnant dans le cadre du SIMH.

Un peu de terminologie : Chaque simulateur se compose d'un *ensemble* standard *de contrôle du simulateur* (SCP et bibliothèques associées), qui fournit un cadre de contrôle et des routines utilitaires pour un simulateur, et d'une *machine virtuelle* (VM) unique, qui met en œuvre le processeur simulé et les périphériques sélectionnés. Une VM se compose de plusieurs *dispositifs*, tels que l'unité centrale, le lecteur de bande papier, le contrôleur de disque, etc. Chaque contrôleur se compose d'un espace d'état nommé (appelé *registres*) et d'une ou plusieurs *unités*. Chaque unité se compose d'un espace d'état numéroté (appelé *ensemble de données*). L'*ordinateur hôte* est le système sur lequel fonctionne le SIMH ; l'*ordinateur cible* est le système simulé.

Le SIMH est ouvertement basé sur le système de simulation MIMIC, conçu à la fin des années 1960 par Len Fehskens, Mike McCarthy et Bob Supnik. Ce document est basé sur la spécification d'interface publiée par MIMIC, "How to Write a Virtual Machine for the MIMIC Simulation System", par Len Fehskens et Bob Supnik.

2. Données Types

Le SIMH est écrit en C. Le système hôte doit supporter (au) les types de données 32 bits (64 bits pour PDP-10 et d'autres systèmes cibles à grand nombre de mots). Pour faire face aux caprices des types de données en C, le SIMH définit des types de données non ambigus pour ses interfaces :

Type de données SIMH	interprétation en langage C 32 bits typique
int8, uint8	char signé, char non signé
int16, uint16	signed short, unsigned short
int32, uint32	signed int, unsigned int
t_int64, t_uint64	long long, _int64 (spécifique au système)
t_addr	adresse simulée, uint32 ou t_uint64
t_value	valeur simulée, uint32 ou t_uint64
t_svalue	valeur signée simulée, int32 ou t_int64
t_mtrec	longueur de l'enregistrement de la bande magnétique, uint32
t_stat	code d'état, int
t_bool	valeur vrai/faux, int

[L'incohérence dans la dénomination de t_int64 et t_uint64 est due à Microsoft VC++, qui utilise int64 comme membre du nom de la structure dans le fichier principal de définitions de Windows].

En outre, le SIMH définit des structures pour chacun de ses principaux éléments de données :

DISPOSITIF	structure de définition du dispositif
UNITÉ	définition de l'unité structure
REG	structure de définition des registres
MTAB	structure de définition des modificateurs

CTAB	structure de définition des commandes
DEBTAB	structure d'entrée de la table de débogage

3. VM Organisation

Une machine virtuelle (VM) est un ensemble de dispositifs liés entre eux par leur logique interne. Chaque périphérique est nommé et correspond plus ou moins à un morceau de matériel sur la machine réelle ; par exemple :

Dispositif VM	Matériel de la machine réelle
CPU	processeur central et mémoire principale
PTR	contrôleur de lecteur de bande de papier et lecteur de bande de papier
TTI	clavier de console
TTO	sortie de la console
DKP	contrôleur et unités de disques

Il peut y avoir plus d'un dispositif par entité matérielle physique, comme pour la console, mais il doit y en avoir au moins un pour chaque dispositif accessible à l'utilisateur. L'un de ces dispositifs aura la prééminence sur les autres.
la responsabilité de diriger les opérations simulées. Normalement, il 'agit de l'unité centrale, mais il peut s'agir d'une entité de niveau supérieur, telle qu'un maître de bus.

La VM s'exécute en fait comme un sous-programme de l'ensemble de contrôle du simulateur (SCP). Il fournit une routine principale pour l'exécution de programmes simulés et d'autres routines et structures de données pour mettre en œuvre les fonctions de commande et de contrôle du SCP. Les interfaces entre une VM et un SCP sont relativement peu nombreuses :

Interface	Fonction
char **sim_name[]**	chaîne de nom du simulateur
REG ***sim_pc**	pointeur sur le compteur de programme simulé
int32 **sim_emax**	nombre maximum de mots dans une instruction
DEVICE *sim_devices**[]**	tableau de pointeurs vers les dispositifs simulés, terminé par
NULL char *sim_stop_messages**[]**	tableau de pointeurs vers les messages d'erreur
t_stat **sim_load** (...)	sous-programme de chargement binaire
t_stat **sim_inst** (void)	sous-programme d'exécution
d'instruction t_stat **parse_sym** (...)	instruction symbolique parse subroutine
t_stat **fprint_sym** (...)	sous-programme d'impression
d'instructions symboliques	

En outre, il existe six interfaces optionnelles, qui peuvent être utilisées dans des situations particulières, telles que les implémentations d'interfaces graphiques :

Interface	Fonction
void (***sim_vm_init**) (void)	pointeur vers la routine d'initialisation de la VM
t_addr (***sim_vm_parse_addr**) (...)	pointeur sur la routine d'analyse d'adresse
void (***sim_vm_fprint_addr**) (...)	pointeur vers la routine de sortie
d'adresse char (***sim_vm_read**) (...)	pointeur sur la routine d'entrée des
commandes	
void (***sim_vm_post**) (...)	pointeur sur la routine de post-traitement des
commandes CTAB ***sim_vm_cmd**	pointeur sur la table des commandes spécifiques
au simulateur	

Il n'y a pas d'organisation obligatoire pour le code VM. La convention suivante a été utilisée jusqu'à . Soit nom le *nom* du système réel (i1401 pour l'IBM 1401 ; i1620 pour l'IBM 1620 ; pdp1 pour le PDP-1 ; pdp18b pour les autres PDP 18 bits ; pdp8 pour le PDP-8 ; pdp11 pour le PDP-11 ; nova pour Nova ; hp2100 pour le HP 21XX ; h316 pour le Honeywell 315/516 ; gri pour le GRI-909 ; pdp10 pour le PDP-10 ; vax pour le VAX ; sds pour le SDS-940) :

- *name*.h contient des définitions pour le simulateur en question
- *name_sys*.c contient toutes les interfaces SCP à l'exception du simulateur d'instructions
- *name_cpu*.c contient le simulateur d'instructions et les structures de données du CPU
- *name_stddev*.c contient les périphériques standard du système réel.
- *name_lp*.c contient l'imprimante de ligne.
- *name_mt*.c contient le contrôleur de bande magnétique, les lecteurs, etc.

Les définitions standard du SIMH se trouvent dans sim_defs.h. Les éléments de base du SIMH sont les suivants :

Module source	fichier d'en-tête	module
scp.c	scp.h	paquet de contrôle
sim_console.c	sim_console.h	bibliothèque d'E/S du terminal sim_fio.c
sim_fio.h	bibliothèque d'E/S de fichiers	
sim_timer.c	sim_timer.h	bibliothèque timer
sim_sock.c	sim_sock.h	bibliothèque d'E/S de sockets sim_ether.c
sim_ether.h	Bibliothèque d'E/S Ethernet	
sim_tmxr.c	sim_tmxr.h	bibliothèque de simulation de multiplexeurs de terminaux sim_tape.c
sim_tape.h	bibliothèque de simulation magtape	

3.1 CPU Organisation

La plupart des unités centrales de traitement remplissent au moins les fonctions suivantes :

- Respect du temps
- Recherche d'instructions
- Décodage d'adresses
- Exécution d'instructions non E/S
- Traitement des commandes E/S
- Traitement des interruptions

L'exécution des instructions est en fait la partie la moins compliquée de la conception ; l'organisation de la mémoire et des E/S doit être abordée en premier.

3.1.1 Temps Base

Afin de simuler des événements asynchrones, tels que l'achèvement des E/S, la VM doit définir et conserver une base temporelle. Celle-ci peut être précise (par , les nanosecondes d'exécution) ou arbitraire (par , le nombre dinstructions exécutées), mais elle doit être utilisée de manière cohérente dans toute la VM. Toutes les VM existantes comptent le temps en instructions.

L'unité centrale est responsable du décompte du compteur d'événements **sim_interval** et de l'appel au contrôleur d'événements asynchrones **sim_process_event**. Le SCP s'occupe de l'enregistrement des données pour le chronométrage.

3.1.2 Étape Fonction

SCP met en œuvre une fonction de pas à l'aide de la commande step. STEP compte un nombre spécifié d'unités de temps (comme décrit dans la section 3.1.1) et arrête ensuite la simulation. La VM peut annuler le décompte de la commande STEP en appelant la routine **sim_cancel_step** :

- t_stat sim_cancel_step (void) - annule le décompte de l'étape.

La VM peut alors inspecter la variable **sim_step** pour voir si une commande STEP est en cours. Si **sim_step** est différent de zéro, il représente le nombre d'étapes à exécuter. La VM peut décompter **sim_step** en utilisant sa propre méthode de comptage, comme les cycles, les instructions ou les références mémoire.

3.1.3 Mémoire Organisation

Le critère pour la disposition de la mémoire est très simple : utiliser le type de données SIMH qui est aussi grand (ou, si nécessaire, plus grand que) la longueur de mot de la machine réelle. Notez que le critère est la longueur du mot, et non l'adressabilité : le PDP-11 a une mémoire adressable par octet, mais c'est une machine 16 bits, et sa mémoire est définie comme uint16 M[]. Il peut sembler tentant de définir la mémoire comme une union des types de données int8 et int16, mais cela rendrait la VM résultante dépendante de l'endian. Au lieu de cela, la VM doit être basée sur la taille de mot sous-jacente de la machine réelle, et la manipulation des octets doit être effectuée de manière explicite. Exemples :

Simulateur	taille de la mémoire	déclaration de mémoire
IBM 1620	5 bits	uint8
IBM 1401	7 bits	uint8
PDP-8	12 bits	uint16
PDP-11, Nova	16 bits	uint16
PDP-1	18 bits	uint32
VAX	32 bits	uint32
PDP-10, IBM 7094	36 bits	t_uint64

3.1.4 Interruption Organisation

La conception de la structure d'interruption de la VM est une interaction complexe entre l'efficacité et la fidélité au matériel. Si la structure d'interruption de la VM est trop abstraite, les logiciels pilotés par interruption risquent de ne pas fonctionner. D'autre part, si elle suit trop fidèlement le matériel, elle peut réduire considérablement la vitesse de simulation. Une règle que je peux proposer est de minimiser le coût de la phase d'acquisition des interruptions, même si cela complique l'évaluation (beaucoup moins fréquente) du système d'interruption à la suite d'une opération d'E/S ou d'un événement asynchrone. Il ne faut pas non plus généraliser à l'excès ; même si le matériel réel peut supporter 64 ou 256 dispositifs d'interruption, les simulateurs utiliseront des configurations beaucoup plus petites. Je commencerai par une structure d'interruption simple, puis je proposerai des suggestions de généralisation.

Dans la structure la plus simple, les demandes d'interruption correspondent à des drapeaux de périphérique et sont conservées dans une variable de demande d'interruption, avec un drapeau par bit. L'évaluation de la phase de récupération des interruptions comprend deux étapes : les interruptions sont-elles activées et y a-t-il une interruption en cours ? Si toutes les demandes d'interruption sont conservées sous forme de drapeaux d'un seul bit dans une variable, le test de la phase d'extraction est très rapide :

```
if (int_enable && int_requests) { ...traiter l'interruption... }
```

En effet, le drapeau de validation d'interruption peut devenir le bit le plus élevé de la variable de demande d'interruption, et les deux tests peuvent être combinés :

```
if (int_requests> INT_ENABLE) { ...traiter l'interruption... }
```

La définition ou l'effacement des drapeaux de périphérique définit ou efface directement le drapeau

de demande d'interruption approprié : set : int_requests = int_requests |

```
DEVICE_FLAG ;
clair :    int_requests= int_requests & ~DEVICE_FLAG ;
```

À un niveau de complexité légèrement supérieur, les demandes d'interruption ne correspondent pas directement aux drapeaux de périphérique, mais sont basées sur le masquage des drapeaux de périphérique avec un masque d'activation (ou de désactivation). Il y a maintenant deux variables parallèles : les drapeaux de périphérique et le masque d'activation d'interruption. Le test de la phase d'extraction est maintenant :

If (int_enable && (dev_flags & int_enables)) { ...traiter l'interruption... }

Dans un deuxième , la VM peut conserver une variable de demande d'interruption récapitulative, qui est mise à jour en cas de modification 'un drapeau de périphérique ou de l'activation/désactivation d'une interruption :

enable : int_requests = device_flags & int_enables ;
disable : int_requests= device_flags & ~int_disables ;

Cela simplifie légèrement le test de la phase d'extraction.

À un niveau de complexité encore plus élevé, système d'interruption peut être trop complexe ou trop grand pour être évalué pendant la phase de recherche. Dans ce cas, un indicateur d'interruption en attente est créé et évalué par un appel de sous-programme chaque fois qu'un changement peut se produire (début d'exécution, instruction d'E/S émise, délai d'attente périphérique). Cela simplifie l'évaluation de la phase d'acquisition et isole l'évaluation des interruptions dans une sous-routine commune.

Si cela est nécessaire pour le traitement des interruptions, le dispositif d'interruption ayant la priorité la plus élevée peut être déterminé en balayant la variable de demande d'interruption de haute priorité à basse priorité jusqu'à ce qu'un bit défini soit trouvé. La position de ce bit peut alors être rétrocompilée dans une table afin de déterminer l'adresse ou le vecteur d'interruption du dispositif d'interruption.

3.1.5 I/O Dispatching

La répartition des E/S se fait en quatre étapes :

- Identifiez la commande d'E/S et analysez l'adresse du périphérique.
- Localisez l'appareil sélectionné.
- Décomposer la commande d'E/S en champs standard.
- Appeler le processeur de l'appareil.

L'analyse d'une commande d'E/S est généralement facile. La plupart des systèmes disposent d'une ou plusieurs instructions d'E/S explicites contenant une commande d'E/S et une adresse de périphérique. L'identification d'une référence à l'espace d'E/S fait partie de l'adressage de la mémoire. Cela nécessite généralement de centraliser les lectures et écritures de la mémoire dans des sous-programmes, plutôt que dans du code en ligne.

Une fois qu'une commande d'E/S a été analysée, l'unité centrale doit localiser le sous-programme du périphérique. Le moyen le plus simple est une grande instruction de commutation avec des appels de sous-programmes câblés. Une méthode plus modulaire consiste à faire appel à une table de répartition, avec des entrées NULL représentant les périphériques inexistants ; cela simplifie également la prise en charge des adresses de périphériques modifiables et des périphériques configurables. Avant d'appeler la routine de périphérique, l'unité centrale décompose généralement la commande d'E/S en champs standard. Cela simplifie l'écriture du simulateur de périphérique.

3.1.6 Instruction Exécution

L'exécution des instructions relève de la responsabilité de la sous-routine **sim_instr** de la VM. Elle est appelée depuis le SCP à la suite d'une commande RUN, GO, CONT ou BOOT. Elle commence à exécuter les instructions au PC actuel (**sim_PC** pointe vers son bloc de description de registre) et continue jusqu'à ce qu'elle soit interrompue par une erreur ou un événement externe.

Lorsqu'elle est appelée, l'unité centrale doit tenir compte de tous les changements d'état effectués par l'utilisateur. Par exemple, il peut être nécessaire de réévaluer si une interruption est en cours ou de restaurer l'état fréquemment utilisé dans des variables de registre locales pour des raisons d'efficacité. Le cycle d'acquisition et d'exécution des instructions est généralement structuré comme une boucle contrôlée par une variable d'erreur, par exemple,

raison= 0 ;
do { ... } while (reason == 0) ; ou while (reason== 0) { ... } À

l'intérieur de cette boucle, l'ordre habituel des événements est le suivant :

- Si le compteur d'événements **sim_interval** a atteint zéro, traiter tous les événements chronométrés. Cette opération est effectuée par le sous-programme SCP **sim_process_event**. Comme il s'agit du mécanisme d'interrogation pour les arrêts du processeur générés par l'utilisateur (^E), les erreurs doivent être reconnues immédiatement :

```
if (sim_interval<= 0) {
        if (reason= sim_process_event ()) break ; }
```

- Vérifier les interruptions en cours et les traiter si nécessaire.

- Vérifier s'il existe d'autres événements propres au processeur, tels que l'état d'attente en suspens ou les pièges en suspens.

- Vérifier la présence d'un point d'arrêt d'instruction. SCP dispose d'un système complet de points d'arrêt. Il permet à une VM de définir de nombreux types de points d'arrêt différents. La VM vérifie les points d'arrêt d'exécution (type E) pendant la recherche d'instructions.

- Récupérer l'instruction suivante, incrémenter le PC, éventuellement décoder l'adresse, et distribuer (via une instruction switch) pour exécution.

Quelques lignes directrices pour la mise en œuvre :

- En général, le code doit refléter le matériel simulé. C'est généralement le plus simple et le plus facile à déboguer.

- La machine virtuelle doit fournir des aides au débogage. Les unités centrales existantes fournissent toutes plusieurs points d'arrêt d'instruction, une file d'attente de changement de PC, des arrêts d'erreur en cas d'instructions ou d'opérations non valides, ainsi qu'un examen et une modification symboliques de la mémoire.

3.2 Organisation des périphériques

Les éléments de base d'une VM sont des dispositifs, chacun correspondant approximativement à un morceau de matériel réel. Un dispositif se compose d'un état basé sur des registres et d'une ou plusieurs unités. Ainsi, un sous-système de disque à plusieurs unités est constitué d'un seul dispositif (représentant le matériel du contrôleur réel) et d'une ou plusieurs unités (chacune représentant une seule unité de disque). Parfois, le dispositif et son unité sont la même entité, comme, par exemple, dans le cas d'un lecteur de bande papier. Toutefois, un dispositif physique unique, tel que la console, peut être divisé, pour des raisons de commodité, en dispositifs d'entrée et de sortie distincts.

En général, les unités correspondent à des sources individuelles d'entrée ou de sortie (un transport de bande, un canal A-D). Les unités constituent le support de base pour la synchronisation et l'entrée-sortie des périphériques. À l'exception de la console, tous les périphériques d'E/S sont simulés en tant que fichiers résidents. SCP permet à l'utilisateur d'établir une association explicite entre un fichier résident et une entité matérielle simulée.

Les appareils et les unités ont tous deux un état. Les appareils fonctionnent sur des *registres* qui contiennent des informations sur l'état de l'appareil et, indirectement, sur l'état des unités. Les unités opèrent sur des *ensembles de données*, qui peuvent être considérés comme des instances individuelles d'entrée ou de sortie, telles qu'un paquet de disques ou une bande de papier perforé. Dans un dispositif multi-unités typique, toutes les unités sont identiques et le dispositif effectue des opérations similaires sur chacune d'entre elles, en fonction de celle qui a été sélectionnée par le programme en cours de simulation.

(Remarque : le SIMH, comme le MIMIC, limite les registres aux périphériques. Les registres répliqués, par l'état actuel d'une unité de disque, sont gérés par des tableaux de registres).

Pour chaque niveau structurel, le SIMH définit, et la VM doit fournir, une structure de données correspondante. Les structures **sim_device** correspondent aux dispositifs, les structures **sim_reg** aux registres, et les structures **sim_unit** aux unités. Ces structures sont décrites en détail dans la section 4.

Les principales fonctions d'un périphérique sont les suivantes

- décodage et exécution des commandes
- synchronisation de l'appareil
- la transmission de données.

Le décodage des commandes est assez évident. Au moins une section du module de code périphérique sera consacrée au traitement des directives émises par l'unité centrale. En général, le décodeur de commandes sera responsable de la manipulation des registres et des drapeaux, ainsi que de l'émission ou de l'annulation des demandes d'E/S. La première tâche est facile, mais la seconde nécessite une compréhension approfondie de la synchronisation des périphériques. Le premier est facile, mais le second nécessite une compréhension approfondie de la synchronisation des périphériques.

3.2.1 Dispositif Timing

Le principal problème de la simulation des dispositifs d'E/S est d'imiter les opérations asynchrones dans un environnement de simulation séquentielle. Heureusement, les caractéristiques temporelles de la plupart des périphériques d'E/S ne varient pas en fonction des circonstances externes. La distinction entre les dispositifs dont la temporisation est générée de manière externe (par exemple, le clavier de la console) et ceux dont la temporisation est générée de manière interne (disque, lecteur de bande papier) est cruciale. Avec un dispositif à temporisation externe, il n'y a aucun moyen de savoir quand une opération en cours commencera ou se terminera ; avec un dispositif à temporisation interne, étant donné l'heure à laquelle une opération commence, l'heure de fin peut être calculée.

Pour un dispositif à temporisation interne, le temps écoulé entre le début et la fin d'une opération est appelé temps d'attente. Voici quelques exemples de dispositifs à temporisation interne et de leurs temps d'attente :

PTR (300 caractères/sec)	3,3 msec
PTP (50 caractères/sec)	20 msec
CLK (fréquence de ligne)	16,6 msec
TTO (30 caractères/sec)	33 msec

Les dispositifs de stockage de masse, tels que les disques et les bandes, n'ont pas un temps de réponse fixe, mais un temps de début à fin peut être calculé en fonction de la position actuelle par rapport à la position souhaitée, de l'état de mouvement, etc.

Pour un dispositif à temporisation externe, il n'existe pas de mécanisme portable par lequel une machine virtuelle peut être informée d'un événement externe (par , la frappe d'une touche). Par conséquent, toutes les machines virtuelles actuelles interrogent le clavier, convertissant ainsi le clavier à temporisation externe en un dispositif à temporisation pseudo-interne. Une restriction plus générale est que le SIMH est monotâche. Les opérations threadées doivent être effectuées par interrogation à l'aide du mécanisme de temporisation des unités, soit avec de vraies unités, soit avec de fausses unités créées expressément pour l'interrogation.

SCP fournit les routines de support pour la synchronisation des périphériques. SCP maintient une liste de périphériques (appelés périphériques actifs) qui sont en cours de . Il fournit également des routines pour interroger ou manipuler cette liste (appelée file d'attente active). Enfin, il fournit une routine permettant de vérifier la présence d'unités temporisées et d'exécuter une commande
Action spécifiée par la VM en cas dépassement de délai.

La temporisation du périphérique est réalisée à l'aide de la structure UNIT, décrite dans la section 4. Pour mettre en place une opération temporisée, le périphérique calcule une période d'attente pour une unité et place cette unité dans la file d'attente active. L'unité centrale compte la période d'attente. Lorsque la période d'attente a expiré, **sim_process_event** retire l'unité de la file d'attente active et appelle un sous-programme du périphérique. Un appareil peut également annuler une opération temporisée en cours et demander l'état de la file d'attente. Les sous-programmes de temporisation sont les suivants

- t_stat **sim_activate** (UNIT *uptr, int32 wait). Cette routine place l'unité spécifiée dans la file d'attente active avec la période d'attente spécifiée. Une période d'attente de 0 est légale ; les périodes d'attente négatives provoquent une erreur. Si l'unité est déjà active, la file d'attente active n'est pas modifiée et aucune erreur ne se produit.

- t_stat **sim_cancel** (UNIT *uptr). Cette routine supprime l'unité spécifiée de la file d'attente active. Si l'unité n'est pas dans la file d'attente, aucune erreur ne se produit.

- int32 **sim_is_active** (UNIT *uptr). Cette routine teste si une unité est dans la file d'attente active. Si c'est le cas, la routine renvoie le temps (+1) restant ; si ce n'est pas le cas, la routine renvoie 0.

- double **sim_gtime** (void). Cette routine renvoie le temps écoulé depuis la dernière commande RUN ou BOOT.

- uint32 **sim_grtime** (void). Cette routine renvoie le 32b de poids faible du temps écoulé depuis la dernière commande RUN ou BOOT.

- int32 **sim_qcount** (void). Cette routine renvoie le nombre d'entrées dans la file d'attente de l'horloge.

- t_stat **sim_process_event** (void). Cette routine retire toutes les unités temporisées de la file d'attente active et appelle le sous-programme approprié de l'appareil pour gérer le délai d'attente.

- int32 **sim_interval**. Cette variable compte à rebours le premier événement chronométré en cours. S'il n'y a pas d'événement chronométré en cours, SCP compte un "intervalle nul" de 10 000 unités de temps.

3.2.2 Horloge Calibration

Le mécanisme de synchronisation décrit dans la section précédente est approximatif. Les dispositifs, tels que les horloges en temps réel, qui suivent l'heure murale seront imprécis. SCP fournit des routines pour synchroniser plusieurs horloges simulées (jusqu'à un maximum de 8) avec l'heure murale.

- int32 **sim_rtcn_init** (int32 clock_interval, int32 clk). Cette routine initialise le mécanisme de calibration de l'horloge pour l'horloge simulée *clk*. L'argument est retourné comme résultat.

- int32 **sim_rtcn_calb** (int32 tickspersecond, int32 clk). Cette routine permet de calibrer l'horloge simulée *clk*. L'argument est le nombre de ticks d'horloge attendus par seconde.

La VM doit appeler **sim_rtcn_init** pour chaque horloge simulée à deux endroits : dans le prologue de **sim_instr**, avant le début de l'exécution des instructions, et à chaque fois que l'horloge temps réel est démarrée. Le simulateur appelle **sim_rtcn_calb** pour calculer le délai d'intervalle réel lorsque l'horloge temps réel est entretenue :

```
/* démarrage de l'horloge */

if (!sim_is_active (&clk_unit)) sim_activate (&clk_unit, sim_rtcn_init (clk_delay, clkno)) ; etc.

/* service d'horloge */

sim_activate (&clk_unit, sim_rtcb_calb (clk_ticks_per_second, clkno) ;
```

L'horloge en temps réel est généralement l'horloge simulée 0 ; d'autres horloges sont utilisées pour l'interrogation des multiplexeurs asynchrones ou des temporisateurs d'intervalles.

3.2.3 Marche au ralenti

Si une VM implémente une horloge calibrée de 100Hz ou moins fonctionnant librement, elle peut également implémenter la marche au ralenti. Le ralenti est un moyen d'interrompre la simulation lorsqu'il n' a pas de travail réel, sans perdre l'étalonnage de l'horloge. La VM doit détecter quand elle est inactive ; elle peut alors informer l'hôte de cette situation en appelant **sim_idle** :

- t_bool **sim_idle** (int32 clk, t_bool one_tick) - tente de mettre la VM au repos jusqu'au prochain événement d'E/S programmé, en utilisant l'horloge simulée *clk* comme base de temps, et décrémente **sim_interval** d'une valeur appropriée.

nombre cycles. Si l'on ne dispose pas d'une minuterie calibrée, ou si le délai avant le prochain événement est inférieur à 1ms, décrémenter **sim_interval** de 1 si *one_tick* est VRAI ; dans le cas contraire, laisser sim_interval inchangé.

sim_idle renvoie TRUE si la VM a réellement tourné au ralenti, FALSE si ce n'est pas le cas.

Comme la marche au ralenti et l'étranglement s'excluent mutuellement, la VM doit informer SCP de l'activation ou de la désactivation de la marche au ralenti :

- t_stat **sim_set_idle** (UNIT *uptr, int32 val, char *cptr, void *desc) - informe le SCP que la marche au ralenti est activée.
- t_stat **sim_clr_idle** (UNIT *uptr, int32 val, char *cptr, void *desc) - informe le SCP que la marche au ralenti est désactivée.
- t_stat **sim_show_idle** (FILE *st, UNIT *uptr, int32 val, void *desc) - affiche si la marche au ralenti est activée ou désactivée, comme le voit SCP.

3.2.4 Données I/O

Pour la plupart des dispositifs, la synchronisation représente la moitié de la bataille (pour les horloges, c'est toute la guerre) ; l'autre moitié est l'E/S. Certains dispositifs sont simulés sur du matériel réel (par , les contrôleurs Ethernet). La plupart des périphériques d'E/S sont simulés sous forme de fichiers sur le système de fichiers de l'hôte au format little-endian. SCP permet d'associer des fichiers à des unités (commande ATTACH) et de lire et d'écrire des données depuis et vers des périphériques dans un format endian et
de manière indépendante de la taille.

Pour la plupart des dispositifs, le concepteur de la VM n'a pas à se préoccuper du formatage des fichiers de dispositifs simulés. Les E/S se font par quantités de 1, 2, 4 ou 8 octets ; SCP choisit automatiquement la taille correcte des données et corrige l'ordre des octets. Problèmes spécifiques :

- Les imprimantes à lignes doivent écrire les données en ASCII 7 bits, les nouvelles lignes remplaçant les séquences de retour chariot et de saut de ligne.

- Les disques doivent être considérés comme des ensembles de données linéaires, du secteur 0 de la surface 0 du cylindre 0 au dernier secteur du disque. Cela permet de transcrire facilement les disques réels en fichiers utilisables par le simulateur.

- Par convention, les bandes magnétiques utilisent un format basé sur des enregistrements. Chaque enregistrement se compose d'une longueur d'enregistrement de 32 bits, des données de l'enregistrement (complétées par un octet de 0 si la longueur de l'enregistrement est impaire) et d'une longueur d'enregistrement de 32 bits. Les marques de fichier sont enregistrées sous la forme d'une longueur d'enregistrement de 0.

- Les cartes ont 12 bits de données par colonne, mais les données sont plus facilement visualisées sous forme de caractères (ASCII). La colonne binaire peut être mise en œuvre en utilisant deux caractères successifs par colonne de carte...

Les E/S de données varient entre les dispositifs à capacité fixe et variable, et entre les dispositifs avec ou sans mémoire tampon. Un dispositif à capacité fixe diffère d'un dispositif à capacité variable en ce que le fichier attaché au premier a une taille maximale, tandis que le fichier attaché au second peut s'étendre indéfiniment. Un périphérique à mémoire tampon diffère d'un périphérique sans mémoire tampon en ce que le premier met en mémoire tampon son ensemble de données dans la mémoire de l'hôte, tandis que le second le conserve sous la forme d'un fichier. La plupart des périphériques à capacité variable (tels que le lecteur de bande de papier et le perforateur) sont séquentiels ; tous les périphériques à mémoire tampon sont à capacité fixe.

3.2.4.1 Lecture et écriture des données

La commande ATTACH crée une association entre un fichier hôte et une unité d'entrée/sortie. Pour les périphériques non tamponnés, ATTACH stocke le pointeur du fichier hôte dans le champ **fileref** de la structure UNIT. Pour les périphériques à mémoire tampon, ATTACH lit l'intégralité du fichier hôte dans un tampon indiqué par le champ **filebuf** de la structure UNIT. Si l'indicateur d'unité UNIT_MUSTBUF est activé, la mémoire tampon est allouée dynamiquement ; sinon, elle doit être allouée statiquement.

Pour les périphériques non tamponnés, les E/S sont effectuées à l'aide de sous-routines C standard et des routines SCP **sim_fread** et **sim_fwrite**. **sim_fread** et **sim_fwrite** sont identiques, en termes de séquence d'appel et de fonction, à fread et fwrite, respectivement, mais ils corrigent les dépendances endian. Pour les périphériques à mémoire tampon, les E/S sont effectuées en copiant les données vers ou depuis la mémoire tampon allouée. Le code du dispositif doit conserver le numéro (+1) de l'adresse la plus élevée modifiée dans le champ **hwmark** de la structure UNIT. Dans les cas non tamponnés et tamponnés, le dispositif doit effectuer tous les calculs d'adresse et toutes les opérations de positionnement.

Le SIMH permet d'accéder à des fichiers d'une taille supérieure à 2 Go (limite de la position int32). Si une VM est compilée avec les drapeaux USE_INT64 et USE_ADDR64 définis, alors t_addr est défini comme t_uint64 au lieu de uint32. La routine sim_fseek permet aux périphériques simulés d'effectuer des accès aléatoires dans de grands fichiers :

- **int sim_fseek** (FILE *handle, t_addr position, int where)

sim_fseek est identique au **fseek** C standard, à deux exceptions près := SEEK_END n'est pas supporté, et l'argument de la position peut avoir une largeur de 64b.

La commande DETACH rompt l'association entre un fichier hôte et une unité d'entrée/sortie. Pour les périphériques à mémoire tampon, DETACH réécrit la mémoire tampon allouée dans le fichier hôte.

3.2.4.2 Console E/S

SCP fournit trois routines pour les entrées/sorties de la console.

- t_stat **sim_poll_char** (void). Cette routine interroge les entrées clavier. S'il y a un caractère, elle renvoie SCPE_KFLAG + le caractère. Si l'utilisateur a tapé le caractère d'interruption (^E), elle renvoie SCPE_STOP. Si la console est reliée à une connexion Telnet et que la connexion est perdue, la routine renvoie SCPE_LOST. S'il n'y a pas d'entrée, la routine renvoie SCPE_OK.

- t_stat **sim_putchar** (int32 char). Cette routine tape le caractère ASCII spécifié sur la console. Si la console est attachée à une connexion Telnet, et que la connexion est perdue, la routine renvoie SCPE_LOST.

- t_stat **sim_putchar_s** (int32 char). Cette routine affiche le caractère ASCII spécifié sur la console. Si la console est attachée à une connexion Telnet, et que la connexion est perdue, la routine renvoie SCPE_LOST ; si la connexion est en attente, la routine renvoie SCPE_STALL.

4. Données Structures

Les dispositifs, les unités et les registres qui composent une VM sont formellement décrits par un ensemble de structures de données qui interfacent la VM avec les parties de contrôle de SCP. Les dispositifs eux-mêmes sont désignés par le tableau de la liste des **dispositifs sim_devices[]**. Au sein d'un dispositif, les unités et les registres sont alloués de manière contiguë sous forme de tableaux de structures. En outre, de nombreux dispositifs permettent à l'utilisateur de définir ou d'effacer des options par le biais d'un tableau de modifications.

Notez qu'un dispositif doit toujours avoir au moins une unité, même si cette unité n'est pas nécessaire à des fins de simulation. Un dispositif doit toujours pointer vers une table de registres valide, mais la table de registres peut se limiter à l'entrée "fin de table".

4.1 sim_device Structure

Les appareils sont définis par la structure **sim_device** (typedef **DEVICE**) :

```
struct sim_device {
    char            *nom ;                          /* nom */
```

```
struct sim_unit *units ;                /* unités */
struct sim_reg    *registers ;          /* registres */
struct sim_mtab *modifiers ;            /* modificateurs */
int32             numunits ;            /* #unités */
uint32            aradix ;              /* radix d'adresse */
uint32            awidth ;              /* largeur de l'adresse */
uint32            aincr ;               /* incrémentation de l'adresse */
uint32            dradix ;              /* radix des données */
uint32            dwidth ;              /* largeur des données */
t_stat            (*examine)() ;        /* examiner la routine */
t_stat            (*dépôt)() ;          /* routine de dépôt */
t_stat            (*reset)() ;          /* routine de réinitialisation */
t_stat            (*boot)() ;           /* routine de démarrage */
t_stat            (*attach)() ;         /* routine d'attachement */
t_stat            (*detach)() ;         /* routine de détachement */
void              *ctxt                 /* contexte */
uint32            drapeaux ;            /* drapeaux */
uint32            dctrl ;               /* drapeaux de contrôle de débogage */
struct sim_debtab debflags ;            /* noms des drapeaux de débogage */
t_stat            (*msize)() ;          /* modification de la taille de la mémoire */
char              *nom ;                /* nom logique */
} ;
```

Les champs sont les suivants :

nom Nom du dispositif, chaîne de caractères alphanumériques majuscules. **units** pointeur sur un tableau de structures **sim_unit**, ou NULL s'il n'y en a pas. **registers** pointeur sur un tableau de structures **sim_reg**, ou NULL s'il n'y en a pas. **modifiers** pointeur sur un **tableau de structures sim_mtab**, ou NULL s'il n'y en a pas. **numunits** nombre d'unités dans ce dispositif.

aradix radix pour la saisie et l'affichage des adresses des appareils, de 2 à 16 inclus.

largeur largeur en bits de l'adresse d'un appareil, de 1 à 64 inclus.

aincr incrément entre les adresses des dispositifs, normalement 1 ; cependant, les dispositifs adressés par octet avec des mots de 16 bits spécifient 2, avec des mots de 32 bits 4.

dradix radix pour l'entrée et l'affichage des données de l'appareil, de 2 à 16 inclus.

largeur largeur en bits des données du dispositif, de 1 à 64 inclus.

examiner adresse de la routine de lecture des données du dispositif spécial, ou NULL si aucune n'est requise. **deposit** adresse de la routine d'écriture des données du dispositif spécial, ou NULL si aucune n'est . **reset** adresse de la routine de réinitialisation du dispositif, ou NULL si aucune n'est requise.

boot adresse de la routine d'amorçage du périphérique, ou NULL si aucune n'est requise. **attach** adresse de la routine d'attachement du périphérique, ou NULL si aucune n'est requise. **detach** adresse de la routine spéciale de déconnexion du périphérique, ou NULL si aucune n'est requise.

ctxt adresse de la table de contexte du périphérique spécifique à la VM, ou NULL si aucune table n'est requise.

drapeaux drapeaux de l'appareil.

dctrl drapeaux de contrôle de débogage.

debflags pointeur sur un tableau de structures sim_debtab, ou NULL s'il n'y en a pas.

msize adresse de la routine de modification de la taille de la mémoire, ou NULL si aucune n'est requise.

lname pointeur sur la chaîne de noms logiques.

4.1.1 Awidth et Aincr

Le champ **awidth** spécifie la largeur du "mot" informatique fondamental de la VM. Par exemple, sur le PDP-11, la **largeur** est de 16b, même si la mémoire est adressable par octets. Le champ **aincr** indique le nombre d'unités d'adressage qui composent le "mot" fondamental. Par exemple, sur le PDP-11, **aincr** est égal à 2 (2 octets par mot).

Si **aincr** est supérieur à 1, SCP suppose que les données sont naturellement alignées sur des adresses qui sont des multiples de
aincr. Les machines virtuelles qui prennent en charge l'alignement arbitraire des données par octet (comme le VAX) peuvent suivre l'une des deux stratégies suivantes :

- Définissez **awidth=** 8 et **aincr=** 1 et ne prenez en charge que l'accès par octet dans les routines d'examen/dépôt.
- Fixer **awidth** et **aincr** aux tailles fondamentales et prendre en charge l'accès aux données non alignées dans routines d'examen/dépôt.

Dans une VM adressable par octet, SAVE et RESTORE nécessitent (memory_size_bytes / **aincr**) itérations pour sauvegarder ou restaurer la mémoire. Il est donc nettement plus efficace d'utiliser une mémoire à l'échelle du mot plutôt qu'à l'échelle de l'octet ; mais les exigences relatives à l'accès non aligné peuvent accroître considérablement la complexité des routines d'examen et de dépôt.

4.1.2 Dispositif Drapeaux

Le champ **flags** contient des indicateurs de l'état actuel de l'appareil. Le SIMH définit 2

indicateurs : nom de l'indicateur signification s'il est activé

DEV_DISABLE l'appareil peut être activé ou désactivé
DEV_DIS l'appareil est actuellement désactivé
DEV_DYNM le dispositif nécessite un appel à la routine **msize** pour modifier la taille de la mémoire DEV_NET le dispositif s'attache au réseau plutôt qu'à un fichier
DEV_DEBUG le périphérique supporte la commande SET DEBUG
DEV_RAW le périphérique prend en charge les E/S brutes DEV_RAWONLY le dispositif ne supporte que les E/S brutes

À partir de la position de bit DEV_V_UF, les autres indicateurs sont spécifiques à l'appareil. Les indicateurs de périphérique sont automatiquement sauvegardés et restaurés ; le périphérique n'a pas besoin de fournir un registre pour ces bits.

4.1.3 Contexte

Ce champ contient un pointeur vers une table de contexte de périphérique spécifique à la VM, si nécessaire. Le SIMH n'accède jamais à ce champ. Le champ contextuel permet au code spécifique à la VM de parcourir les structures de données spécifiques à la VM à partir du pointeur racine **sim_devices.**

4.1.4 Examiner et déposer Routines

Pour les dispositifs qui gèrent leurs ensembles de données comme des fichiers hôtes, SCP met en œuvre les fonctions d'examen et de dépôt des données. Cependant, les périphériques qui gèrent leurs ensembles de données en tant qu'état privé (par , l'unité centrale) doivent fournir des routines spéciales d'examen et de dépôt. Les séquences d'appel sont les suivantes :

t_stat *examine_routine* (t_val *eval_array, t_addr addr, UNIT *uptr, int32 switches) - Copie les adresses consécutives **sim_emax** pour l'unité *uptr*, en commençant par *addr*, dans *eval_array*. La variable *switch* a le bit<n> positionné si la n'ième lettre a été spécifiée comme switch à la commande examine.

t_stat *deposit_routine* (t_val value, t_addr addr, UNIT *uptr, int32 switches) - Stocke l'information spécifiée dans la base de données.
dans l'*addr* spécifié pour l'unité *uptr*. La variable de *commutation* est la même que pour la routine examine.

4.1.5 Réinitialiser Routine

La routine de réinitialisation met en œuvre la fonction de réinitialisation du dispositif pour les commandes RESET, RUN et BOOT. Sa séquence d'appel est la suivante :

t_stat *reset_routine* (DEVICE *dptr) - Remet le périphérique spécifié dans son état initial.

Une routine de réinitialisation typique efface tous les drapeaux de l'appareil et annule toutes les opérations de synchronisation en cours. Le commutateur -p spécifie une réinitialisation à l'état de mise sous tension.

4.1.6 Boot Routine

Si un appareil répond à une commande BOOT, la routine d'amorçage met en œuvre la fonction d'amorçage. Sa séquence d'appel est la suivante :

t_stat *boot_routine* (int32 unit_num, DEVICE *dptr) - Amorcer l'unité *unit_num* sur le périphérique *dptr*.

Une routine d'amorçage typique copie un chargeur d'amorçage dans la mémoire principale et place le PC à l'adresse de départ du chargeur. SCP démarre alors la simulation à l'adresse spécifiée.

4.1.7 Routines d'attachement et de détachement

Normalement, les commandes ATTACH et DETACH sont traitées par SCP. Toutefois, les dispositifs qui pré ou post-traiter ces commandes doivent fournir des routines d'attachement et de détachement spéciales. Les séquences d'appel sont les suivantes :

t_stat *attach_routine* (UNIT *uptr, char *file) - Attache le *fichier* spécifié à l'unité *uptr*. Sim_switches contient le commutateur de commande ; le bit SIM_SW_REST indique que attach est appelé par la commande RESTORE plutôt que par la commande ATTACH.

t_stat *detach_routine* (UNIT *uptr) - Détachement de l'unité *uptr*.

En pratique, ces routines invoquent généralement les routines SCP standard, **attach_unit** et **detach_unit**, respectivement. Par exemple, voici des routines spéciales d'attachement et de détachement pour mettre à jour l'état d'erreur de l'imprimante de ligne :

```
t_stat lpt_attach (UNIT *uptr, char *cptr) { t_stat r
       ;
       if ((r= attach_unit (uptr, cptr)) != SCPE_OK) return r ;
       lpt_error = 0 ;
       retourner SCPE_OK ;
       }

t_stat lpt_detach (UNIT *uptr) { lpt_error
       = 1 ;
       return detach_unit (uptr) ;
       }
```

Si la VM spécifie une routine ATTACH ou DETACH, SCP contourne ses tests normaux avant d'appeler la routine VM. Ainsi, une routine VM DETACH ne peut pas être assurée que l'unité est réellement attachée et doit tester les drapeaux de l'unité si nécessaire.

SCP exécute une commande DETACH ALL dans le cadre de la sortie du simulateur. Normalement, DETACH ALL n'appelle la routine de détachement d'une unité que si l'indicateur UNIT_ATT de l'unité est activé. Lors de la sortie du simulateur, la routine de détachement est également appelée si l'unité n'est pas marquée comme attachable (UNIT_ATTABLE n'est pas activé). Cela permet à la routine de détachement d'une unité non attachable de fonctionner comme une routine de nettoyage spécifique au simulateur pour l'unité, le dispositif ou le simulateur entier.

4.1.8 Changement de la taille de la mémoire Routine

La plupart des unités instancient tout tableau de mémoire à la taille maximale possible. Cela permet de modifier la taille apparente de la mémoire en changeant le champ **capac** dans la structure de l'unité. Pour certains appareils (comme l'unité centrale VAX), l'instanciation de la taille maximale de la mémoire imposerait une charge de ressources importante si moins de mémoire était réellement nécessaire. Ces dispositifs doivent fournir une routine, la routine de modification de la taille de la mémoire, que RESTORE doit utiliser si la taille de la mémoire doit être modifiée :

> t_stat *change_mem_size* (UNIT *uptr, int32 val, char *cptr, void *desc) - Change la capacité (taille de la mémoire) de l'unité *uptr* en *val*. Les arguments *cptr* et *desc* sont inclus pour des raisons de compatibilité avec la séquence d'appel de la routine de validation de la commande SET.

4.1.9 Debug Contrôles

Les périphériques peuvent prendre en charge les impressions de débogage. Les impressions de débogage sont contrôlées par la commande SET {NO}DEBUG, qui spécifie où la sortie de débogage doit être imprimée, et par la commande SET <device> {NO}DEBUG, qui active ou désactive les impressions de débogage individuelles.

Si un périphérique prend en charge les impressions de débogage, l'indicateur DEV_DEBUG doit être activé. Le champ **dctrl** est utilisé pour les drapeaux de contrôle de débogage. Si un périphérique ne prend en charge qu'un seul drapeau de débogage, le champ **debflags** doit mis à NULL. Si un périphérique supporte plusieurs drapeaux de débogage, la correspondance entre les positions des bits dans **dctrl** et les noms des drapeaux de débogage est spécifiée dans le tableau **debflags. debflags** pointe vers un tableau continu de structures **sim_debtab** (typedef **DEBTAB**). Chaque structure sim_debtab spécifie un seul drapeau de débogage :

```
Structure sim_debtab {
    char        nom ;          /* nom du drapeau */
    uint32      masque ;       /* bit de contrôle */
   } ;
```

Les champs sont les suivants :

nom	nom de l'indicateur de débogage.
masque	masque de bits du drapeau de débogage.

Le tableau se termine par une entrée NULL.

4.2 sim_unit Structure

Les unités sont allouées sous forme de tableau contigu. Chaque unité est définie par une structure **sim_unit** (typedef

UNIT) : struct sim_unit {

```
    struct sim_unit   *suivant ;      /* prochain actif */
    t_stat            (*action)() ;   /* routine d'action */
    char              *filename ;     /* nom du fichier ouvert
                                         */
    DOSSIER           *fileref ;      /* référence au fichier
                                         */
    vide              *filebuf ;      /* mémoire tampon */
    uint32            hwmark ;        /* niveau de la nappe
                                         phréatique */
    int32             temps ;         /* time out */
    uint32            drapeaux ;      /* drapeaux */
    t_addr            capacité ;      /* capacité */
    t_addr            pos ;           /* position du fichier */
    int32             buf ;           /* tampon */
    int32             attendre ;      /* wait */
    int32             u3 ;            /* spécifique à
                                         l'appareil */
    int32             u4 ;            /* spécifique à
                                         l'appareil */
```

```
int32       u5 ;        /* spécifique à l'appareil */
int32       u6 ;        /* spécifique à l'appareil */
} ;
```

Les champs sont les suivants :

suivant	pointeur sur l'unité suivante dans la file d'attente active, NULL s'il n'y en a pas.
action	adresse de la routine de service de dépassement de délai de l'unité.
nom du fichier	pointeur sur le nom du fichier joint, NULL s'il n'y en a pas.
fileref	pointeur sur la structure FILE du fichier attaché, NULL s'il n'y en a pas.
hwmark	dispositifs tamponnés uniquement ; adresse modifiée la plus élevée,+ 1.
temps	incrémenter jusqu'à ce que le délai dépasse l'unité précédente dans la file d'attente active.
drapeaux	les drapeaux de l'unité.
capac	capacité de l'unité, 0 si variable.
pos	dispositifs séquentiels uniquement ; adresse du dispositif suivant à lire ou à
écrire. **buf**	par convention, le tampon de l'unité, mais peut être utilisé à d'autres fins.
wait	par convention, le temps d'attente de l'unité, mais peut être utilisé à d'autres
fins. **u3**	défini par l'utilisateur.
u4	défini par l'utilisateur.
u5	défini par l'utilisateur.
u6	défini par l'utilisateur.

buf, wait, u3, u4, u5, u6 et certaines parties des **drapeaux** sont tous sauvegardés et restaurés par les commandes SAVE et RESTORE et peuvent donc être utilisés pour l'état de l'unité qui doit être préservé.

La macro **UDATA** est disponible pour remplir les champs communs d'un UNIT. Elle est

invoquée par UDATA (action_routine, drapeaux, capacité)

Les champs après **buf** peuvent être remplis manuellement, par exemple

```
UNITÉ =
          { UDATA (&lpt_svc, UNIT_SEQ+UNIT_ATTABLE, 0), 500 } ;
```

définit l'imprimante de ligne comme une unité séquentielle avec temps d'attente de 500.

4.2.1 Unité Drapeaux

Le champ **flags** contient des indicateurs de l'état actuel de l'unité. Le SIMH définit 12

indicateurs : nom de l'indicateur Signification s'il est activé

UNIT_ATTABLE	l'unité répond à ATTACH et DETACH. UNIT_RO
	l'unité est actuellement en lecture seule.
UNIX_FIX	l'unité est à capacité fixe.
UNIT_SEQ	l'unité est séquentielle.
UNIT_ATT	l'unité est actuellement attachée à un fichier.
UNIT_BINK	l'unité mesure "K" en 1024 au lieu de 1000.
UNIT_BUFABLE	l'unité met en mémoire tampon son ensemble de
données.	
UNIT_MUSTBUF	l'unité alloue son tampon de données de manière
dynamique. UNIT_BUF	l'unité est en train de mettre en tampon son ensemble
de données. UNIT_ROABLE	l'unité peut être attachée en lecture seule.
UNIT_DISABLE	l'unité répond à ENABLE et DISABLE. UNIT_DIS
	l'unité est actuellement désactivée.
UNIT_RAW	l'unité est attachée en mode RAW.

À partir de la position de bit UNIT_V_UF, les autres indicateurs sont spécifiques à l'unité. Les indicateurs spécifiques à l'unité sont activés et désactivés à l'aide des commandes SET et CLEAR, qui font référence au tableau MTAB (voir ci-dessous). Les indicateurs spécifiques à l'unité et UNIT_DIS sont automatiquement sauvegardés et restaurés ; il n'est pas nécessaire que l'appareil fournisse un registre pour ces bits.

4.2.2 Service Routine

Cette routine est appelée par **sim_process_event** lorsqu'une unité s'arrête. La séquence d'appel est

la suivante : t_stat *service_routine* (UNIT *uptr)

Le statut renvoyé par la routine de service est transmis par **sim_process_event** à l'unité centrale.

4.3 sim_reg Structure

Les registres sont alloués sous forme de tableau contigu, avec un registre NULL à la fin. Chaque registre est défini par une structure **sim_reg** (typedef **REG**) **:**

```
struct reg {
    char        *nom ;          /* nom */
    vide        *loc ;          /* emplacement */
    uint32      radix ;         /* radix */
    uint32      la largeur ;    /* largeur */
    uint32      compenser ;     /* bit de départ */
    uint32      profondeur ;    /* sauvegarde de la
                                   profondeur */
    uint32      drapeaux ;      /* drapeaux */
    uint32      qptr ;          /* pointeur de la file d'attente
                                   actuelle */
    } ;
```

Les champs sont les suivants :

nom	nom de l'appareil, chaîne de caractères alphanumériques majuscules.
loc	pointeur sur l'emplacement de la valeur du registre.
radix	radix pour l'entrée et l'affichage des données, de 2 à 16 inclus.
largeur	largeur en bits des données, de 1 à 32 inclus.
largeur	décalage d'un bit (à partir de la fin droite des données).
profondeur	taille du tableau de données (normalement 1).
drapeaux	les drapeaux et les informations de formatage.
qptr	pour une file d'attente circulaire, le numéro de la première entrée

Le champ de **profondeur** est utilisé avec les "registres en tableau". Les registres en tableau sont utilisés pour représenter des structures comportant plusieurs valeurs de données, telles que les emplacements d'un tampon de transfert, ou des structures répliquées dans chaque unité, telles que le registre d'état d'un lecteur. Le champ **qptr** est utilisé avec les " en file d'attente". Les registres en file d'attente sont des tableaux organisés en files d'attente circulaires, comme la file d'attente des modifications du PC.

Un registre de 32b ou moins conserve ses données dans une variable scalaire de 32b (signée ou non). Un registre de 33b ou plus conserve ses données dans variable scalaire de 64b (signée ou non). Il existe plusieurs exceptions à cette règle :

- Un registre en tableau conserve ses données dans un tableau C dont le type de données SIMH est aussi grand que (ou si nécessaire, plus grand que) la largeur d'un élément de registre. Par exemple, un tableau de registres de 6b conservera ses données dans un tableau uint8 (ou int8) ; un tableau de registres de 16b conservera ses données dans un tableau uint16 (ou int16) ; un tableau de registres de 24b conservera ses données dans un tableau uint32 (ou int32).

- Un registre marqué REG_FIT obéit aux règles de dimensionnement d'un registre en tableau plutôt qu'à celles d'un registre scalaire normal. Ceci est utile pour aliéner des registres dans la mémoire ou dans des structures.

Les macros **ORDATA**, **DRDATA** et **HRDATA** définissent respectivement des registres octal, décimal et hexadécimal justifiés à droite. Elles sont invoquées par :

xRDATA (nom, emplacement, largeur)

La macro **FLDATA** définit un drapeau binaire d'un bit à un décalage arbitraire dans un mot de 32 bits. Elle

est invoquée par : FLDATA (nom, emplacement, position_bite)

La macro **GRDATA** définit un registre avec un emplacement et un radix arbitraires. Elle est

invoquée par : GRDATA (nom, emplacement, radix, largeur, position du bit)

La macro **BRDATA** définit un registre en tableau dont les données sont conservées dans un tableau C standard.

Elle est invoquée par : BRDATA (nom, emplacement, radix, largeur, profondeur)

Pour toutes ces macros, le champ de l'**indicateur** peut être rempli manuellement, par exemple,

```
REG lpt_reg= {
       { DRDATA      (POS, lpt_unit.pos, 31), PV_LFT }, ... }
```

Enfin, la macro **URDATA** définit un registre en tableau dont les données font partie de la structure **UNIT**. Cette macro doit être utilisée avec beaucoup de précautions. Si les champs sont mal configurés ou si les données sont en fait conservées ailleurs, le stockage via cette déclaration de registre peut piétiner la mémoire. La macro est invoquée par :

URDATA (nom, emplacement, radix, largeur, décalage, profondeur, drapeaux)

L'emplacement doit être un décalage dans la structure **UNIT** pour l'unité 0. La largeur doit être de 32 pour un champ int32 ou uint32, et T_ADDR_W pour un champ t_addr. Les drapeaux peuvent être n'importe lequel des drapeaux de registre normaux ; REG_UNIT sera automatiquement intégré par OU. Par exemple, l'exemple suivant déclare un registre en tableau de tous les champs de position **UNIT** dans un appareil avec 4 unités :

```
{ URDATA      (POS, dev_unit[0].pos, 8, T_ADDR_W, 0, 4, 0) }
```

4.3.1 Enregistrer Drapeaux

Le champ **flags** contient des indicateurs qui contrôlent l'examen et le dépôt du registre.

flag name	signification si spécifié
PV_RZRO	registre d'impression justifié à droite avec des zéros en tête. PV_RSPC registre d'impression justifié à droite avec des espaces en tête. PV_LEFT impression du registre justifiée à gauche.
REG_RO	est en lecture seule.
REG_HIDDEN	le registre est caché (il n'apparaîtra pas dans EXAMINE STATE). REG_HRO le registre est en lecture seule et caché.
REG_NZ	les nouvelles valeurs des registres doivent être non nulles.
REG_UNIT	réside dans la structure **UNIT**.
REG_CIRC	est une file d'attente circulaire.
REG_VMIO	est affiché et analysé à l'aide des routines de données VM.
REG_VMAD	est affiché et analysé à l'aide des routines d'adresse VM.

REG_FIT Le conteneur de registre utilise des règles de taille en tableau plutôt que des règles de taille scalaire.

4.4 sim_mtab Structure

Les commandes SHOW et SET spécifiques au dispositif sont traitées à l'aide du tableau des modifications, qui est alloué comme un tableau contigu, avec un NULL à la fin. Chaque modification possible est définie par une structure **sim_mtab** (synonyme **MTAB**), qui comporte les champs suivants :

```
struct sim_mtab {
    uint32      masque ;            /* masque */
    uint32      match ;             /* match */
    char        *pstring ;          /* imprimer la chaîne de
                                       caractères */
    char        *mstring ;          /* chaîne de
                                       correspondance */
    t_stat      (*valide)() ;       /* routine de validation */
    t_stat      (*disp)() ;         /* routine d'affichage */
    vide        *desc ;             /* descripteur
                                       d'emplacement */
    } ;
```

La MTAB prend en charge deux interprétations différentes de la structure : régulière et étendue. Une entrée MTAB normale modifie les drapeaux dans le mot de drapeaux UNIT ; l'entrée du descripteur n'est pas utilisée. Les champs sont les suivants :

masque masque de bits pour tester le champ unit.**flags**
correspondance valeur à mémoriser (SET) ou à comparer (SHOW)
pstring pointeur sur la chaîne de caractères imprimée lors d'une correspondance (SHOW), ou NULL
mstring pointeur sur la chaîne de caractères à comparer (SET), ou NULL
valide adresse de la routine de validation (SET), ou NULL
disp adresse de la routine d'affichage (SHOW), ou

NULL Pour SET, une entrée MTAB normale est interprétée comme suit :

1. Teste si l'entrée **mstring** existe.
2. Teste si le paramètre SET correspond à la **chaîne de caractères**.
3. Appeler la routine de validation, le cas échéant.
4. Appliquer la valeur du **masque** au mot indicateur UNIT et ensuite ou dans la valeur de **correspondance.**

Pour SHOW, une entrée MTAB normale est interprétée comme suit :

1. Teste si l'entrée **pstring** existe.
2. Test pour voir si le mot indicateur UNIT, masqué avec la valeur du **masque**, est égal à la valeur de **correspondance**.
3. Si une routine d'affichage existe, lappeler, sinon
4. Imprimer la **chaîne de caractères pstring**.

Les entrées MTAB étendues sont interprétées différemment :

masque drapeaux d'entrée
MTAB_XTD entrée étendue
MTAB_VDV valable pour les dispositifs MTAB_VUN
valable pour les unités MTAB_VAL prend une valeur
MTAB_NMO valable uniquement dans named SHOW
MTAB_NC ne pas convertir la valeur de l'option en majuscules MTAB_SHP Le paramètre SHOW prend une valeur optionnelle
correspondre valeur à mémoriser (SET)
pstring pointeur sur la chaîne de caractères imprimée lors d'une correspondance (SHOW), ou NULL
mstring pointeur sur la chaîne de caractères à comparer (SET), ou NULL

valide	adresse de la routine de validation (SET), ou NULL
disp	adresse de la routine d'affichage (SHOW), ou
NULL **desc**	pointeur sur une structure REG (MTAB_VAL set)
ou	
	une structure spécifique à la validation (MTAB_VAL

clear) Pour SET, une entrée MTAB étendue est interprétée comme suit :

1. Teste si l'entrée **mstring** existe.
2. Teste si le paramètre SET correspond à la **chaîne de caractères**.
3. Tester si l'entrée est valide pour le type de SET effectué (dispositif SET ou unité SET).
4. S'il existe une routine de validation, il faut l'appeler et renvoyer son état. La routine de validation est responsable du stockage du résultat.
5. Si **desc** est NULL, exit.
6. Si MTAB_VAL est défini, l'option SET est analysée comme "option=n" et la valeur n est stockée dans le registre décrit par **desc**.
7. Dans le cas contraire, la valeur de la **correspondance est** stockée dans l'int32 désigné par **desc**.

Pour SHOW, une entrée MTAB étendue est interprétée comme suit :

1. Teste si l'entrée **pstring** existe.
2. Tester si l'entrée est valide pour le type de SHOW en cours (appareil ou unité).
3. Si une routine d'affichage existe, il faut appeler, sinon,
4. Si MTAB_VAL est défini, imprimer "pstring=n", où la valeur, le radix et la largeur sont tirés du registre décrit par **desc**, sinon,
5. Imprimer la **chaîne de caractères pstring**.

SHOW [dev|unit] <modifier>{=<value>} est un cas particulier. Seuls deux types de modificateurs peuvent être affichés individuellement : une entrée MTAB étendue qui prend une valeur et toute entrée MTAB avec à la fois une routine d'affichage et une **chaîne de caractères**. Rappelons que si une routine d'affichage existe, SHOW n'utilise pas l'entrée **pstring**. Pour l'affichage d'un modificateur nommé, **pstring** est utilisé comme chaîne de correspondance. Cela permet d'implémenter des routines d'affichage complexes qui ne sont invoquées que par leur nom, par exemple,

```
MTAB cpu_tab[]= {
       { masque, valeur, "normal", "NORMAL", NULL, NULL, NULL },
       { MTAB_XTD|MTAB_VDV|MTAB_NMO, 0, "SPECIAL",
              NULL, NULL, NULL, &spec_disp },
       { 0 }
       } ;
```

Une commande SHOW CPU n'affichera que le modificateur nommé NORMAL ; mais SHOW CPU SPECIAL invoquera la routine d'affichage spéciale.

4.4.1 Validation Routine

La routine de validation peut être utilisée pour valider les entrées pendant le traitement SET. Elle peut effectuer d'autres changements d'état requis par la modification ou lancer des dialogues supplémentaires nécessaires au modificateur. Sa séquence d'appel est la suivante :

t_stat *validation_routine* (UNIT *uptr, int32 value, char *cptr, void *desc) - teste que *uptr*.**flags** peut être mis à *value*. *cptr* pointe vers la partie valeur de la chaîne de paramètres (tous les caractères après le signe =) ; si *cptr* est NULL, aucune valeur n'a été donnée. *desc* pointe vers le **REG** ou int32 utilisé pour stocker le paramètre.

4.4.2 Affichage Routine

La routine d'affichage est appelée pendant le traitement de SHOW pour afficher l'état spécifique de l'appareil ou de l'unité. La séquence d'appel est la suivante :

> t_stat *display_routine* (FILE *st, UNIT *uptr, int value, void *desc) - sortie de l'état spécifique au dispositif ou à l'unité pour *uptr* vers le flux *st*. Si le modificateur est une entrée MTAB normale, ou une entrée étendue sans MTAB_SHP, *desc* pointe vers la structure de l'entrée MTAB. Si le modificateur est une entrée MTAB étendue avec , *desc* pointe vers la chaîne de valeur optionnelle ou est NULL si aucune valeur n'a été fournie. *value* est le champ de valeur de l'entrée MTAB correspondante.

Lorsque la routine d'affichage est appelée pour une entrée MTAB normale, SHOW a affiché l'argument **pstring** mais n'a pas ajouté de nouvelle ligne. Lorsqu'elle est appelée pour une entrée MTAB étendue, SHOW n'a rien affiché. SHOW ajoutera une nouvelle ligne après le retour de la routine d'affichage, sauf pour les entrées pour lesquelles le drapeau MTAB_NMO est activé.

4.5 Autres données Structures

char **sim_name[]** est un tableau de caractères contenant le nom de la VM.

int32 **sim_emax** contient le nombre maximal de mots nécessaires pour contenir la plus grande instruction ou donnée de la VM. L'examen et le dépôt traiteront jusqu'à **sim_emax** mots.

DEVICE *sim_devices**[]** est un tableau de pointeurs vers tous les périphériques de la VM. Il est terminé par un NULL. Par convention, le CPU est toujours le premier périphérique du tableau.

REG ***sim_PC*** pointe vers la structure **reg** du compteur de programme. Par convention, le PC est toujours le premier registre du tableau de registres du CPU.

char *sim_stop_messages**[]** est un tableau de pointeurs vers des chaînes de caractères, correspondant à des retours d'état d'erreur supérieurs à zéro. Si **sim_instr** renvoie le code d'état n> 0, alors **sim_stop_message[n]** est imprimé par SCP.

5. Routines fournies par VM

5.1 Instruction Exécution

L'exécution des instructions est effectuée par la routine **sim_instr**. Sa séquence d'appel

> est la suivante : t_stat **sim_instr** (void) - exécution à partir du PC actuel jusqu'à
>
> l'erreur ou l'arrêt.

5.2 Chargement binaire et Dump

Si le VM répond à la commande LOAD (ou DUMP), la routine de chargement (routine de vidage) est mise en œuvre par la routine **sim_load**. Sa séquence d'appel est la suivante :

> t_stat **sim_load** (FILE *fptr, char *buf, char *fnam, t_bool flag) - Si *flag*= 0, chargement des données du fichier binaire *fptr*. Si *flag* = 1, vidage des données dans le fichier binaire *fptr*. Pour l'une ou l'autre commande, *buf* contient tous les arguments spécifiques à la VM et *fnam* contient le nom du fichier.

Si LOAD ou DUMP n'est pas implémenté, **sim_load** doit simplement renvoyer SCPE_ARG. Les commandes LOAD et DUMP ouvrent et ferment le fichier spécifié pour **sim_load**.

5.3 Examen symbolique et Dépôt

Si la VM permet l'examen symbolique et le dépôt de données, elle doit fournir deux routines, **fprint_sym** pour la sortie et **parse_sym** pour l'entrée. Leurs séquences d'appel sont les suivantes :

> t_stat **fprint_sym** (FILE *fichier, t_addr addr, t_value *val, UNIT *uptr, int32 switch) - En fonction de la variable *switch*, envoie symboliquement au flux de données du tableau *val* à l'*addr* spécifié dans l'unité *uptr*.
>
> t_stat **parse_sym** (char *cptr, t_addr addr, UNIT *uptr, t_value *val, int32 switch) - Basé sur la méthode des *switch* variable, analyse la chaîne de caractères *cptr* pour une valeur symbolique *val* à l'*adresse* spécifiée dans l'unité *uptr*.

Si le traitement symbolique n'est pas mis en œuvre, ou si la valeur de sortie ou la chaîne d'entrée ne peut pas être analysée, ces routines doivent renvoyer SCPE_ARG. Si le traitement a réussi et a consommé plus d'un mot, ces routines doivent renvoyer le nombre supplémentaire d'unités d'adressage consommées sous la forme d'un nombre **négatif.** Si le traitement a réussi et a consommé une seule unité d'adressage, ces routines doivent renvoyer SCPE_OK. Par exemple, PDP-11 **parse_sym** répondrait comme suit à diverses entrées :

entrée	valeur de retour
XYZGH	SCPE_ARG
MOV R0,R1	-1
MOV #4,R5	-3
MOV 1234,5670	-5

Il existe une relation implicite entre les arguments **addr** et **val** et les champs **aincr** du périphérique. Chaque entrée de **val** est supposée représenter les unités d'adressage **aincr**, en commençant par **addr** :

val[0]	addr+ 0
val[1]	addr+ aincr
val[2]	addr + (2 * aincr)
val[3]	addr + (3 * aincr)
:	:

Comme **val** est généralement rempli et stocké par des appels aux routines examine et deposit du périphérique, respectivement, les routines examine et deposit et **fprint_sym** et **fparse_sym** doivent se mettre d'accord sur la largeur prévue des éléments dans **val** et sur l'alignement d'**addr**. En outre, si **fparse_sym** souhaite modifier une unité de stockage plus étroite que **awidth**, il doit insérer les nouvelles données dans l'entrée appropriée de **val** sans détruire les champs environnants.

L'interprétation des valeurs des commutateurs est arbitraire, mais les valeurs suivantes sont

utilisées par les machines virtuelles existantes : commutateur	interprétation
-a	un seul caractère
-c	chaîne de caractères
-m	mnémonique d'instruction

En outre, lors de la saisie, un ' (apostrophe) de tête est interprété comme un caractère unique, et un " (guillemet double) de tête est interprété comme une chaîne de caractères.

5.4 Interfaces en option

Pour une plus grande flexibilité, SCP fournit quelques interfaces optionnelles qui peuvent être utilisées pour étendre ses capacités d'entrée, de traitement et de post-traitement des commandes. Ces interfaces sont strictement facultatives

et sont désactivés par défaut. utilisation nécessite une connaissance approfondie du fonctionnement interne de SCP et n'est pas recommandée au rédacteur novice de VM.

5.4.1 Initialisation unique Routine

SCP définit un pointeur (***sim_vm_init**)(void). Il s'agit d'un "global faible" ; si aucun autre module ne définit cette valeur, elle prendra par défaut la valeur NULL. Une VM nécessitant une initialisation spéciale doit remplir ce pointeur avec l'adresse de sa routine d'initialisation spéciale :

```
void sim_special_init (void) ;
void (*sim_vm_init)(void)= &sim_special_init ;
```

La routine d'initialisation spéciale peut effectuer toutes les actions requises par la VM. Si les autres interfaces optionnelles doivent être utilisées, la routine d'initialisation peut remplir les pointeurs appropriés ; cependant, cela peut tout aussi bien fait dans la routine de réinitialisation du processeur.

5.4.2 Entrée d'adresse et affichage

SCP définit un pointeur t_addr *(**sim_vm_parse_addr**)(DEVICE *, char *, char **). Ce pointeur est initialisé à NULL. S'il est rempli par la VM, SCP utilisera la routine spécifiée pour analyser les adresses à la place de sa routine d'entrée numérique standard. La séquence d'appel de la routine **sim_vm_parse_addr** est la suivante :

> t_addr **sim_vm_parse_addr** (DEVICE *dptr, char *cptr, char **optr) - analyse la chaîne indiquée par *cptr* en tant qu'adresse pour le périphérique indiqué par *dptr*. *optr* indique le premier caractère qui n'a pas été analysé avec succès. Si *cptr*== *optr*, l'analyse a échoué.

SCP définit un pointeur void *(**sim_vm_fprint_addr**)(FILE *, DEVICE *, t_addr). Ce pointeur est initialisé à NULL. S'il est rempli par la VM, SCP utilisera la routine spécifiée pour imprimer les adresses à la place de sa routine de sortie numérique standard. La séquence d'appel de la routine **sim_vm_fprint_addr** est la suivante :

> t_addr **sim_vm_fprint_addr** (FILE *stream, DEVICE *dptr, t_addr addr) - adresse de sortie *addr* to dans le format requis par le périphérique indiqué par *dptr*.

5.4.3 Entrée des commandes et traitement post

SCP définit un pointeur char* (**sim_vm_read**)(char *, int32 *, FILE *). Ce pointeur est initialisé à NULL. S'il est rempli par la VM, SCP utilisera la routine spécifiée pour obtenir l'entrée de commande à la place de sa routine standard, read_line. La séquence d'appel de la routine **sim_vm_read** est la suivante :

> char **sim_vm_input** (char *buf, int32 *max, FILE *stream) - lit la ligne de commande suivante à partir de et le stocker dans *buf*, jusqu'*à un maximum de max* caractères

La routine est censée supprimer les caractères d'espacement initiaux et renvoyer NULL à la fin du fichier.

SCP définit un pointeur void *(**sim_vm_post**)(t_bool from_scp). Il est initialisé à NULL. S'il est rempli par la VM, SCP appellera la routine spécifiée à la fin de chaque commande. Cela permet à la VM de mettre à jour tout état local, tel que l'affichage d'une console GUI. La séquence d'appel de la routine vm_post est la suivante :

> void **sim_vm_postupdate** (t_bool from_scp) - si appelé depuis SCP, l'argument *from_scp* est VRAI ; sinon, il est FAUX.

5.4.4 Commandes spécifiques à la VM

SCP définit un pointeur CTAB ***sim_vm_cmd**. Il est initialisé à NULL. S'il est rempli par la VM, SCP l'interprète comme un pointeur vers la table de commandes SCP. Cette table de commandes est vérifiée avant que la saisie de l'utilisateur ne soit recherchée dans la table de commandes standard.

Une table de commande est allouée sous la forme d'un tableau contigu. Chaque entrée est définie par une structure **sim_ctab** (typedef **CTAB**) :

```
struct sim_ctab {
        char        *nom ;              /* nom */
        t_stat      (*action)() ;       /* routine d'action */
        int32       arg ;               /* argument */
        char        *help ;             /* chaîne d'aide */
        } ;
```

Si le premier mot d'une ligne de commande correspond à ctab.name, la routine d'action est appelée avec les arguments suivants :

> t_stat **action_routine** (int32 arg, char *buf) - traite la chaîne d'entrée *buf* en fonction de l'argument optionnel **arg**

La chaîne transmise à la routine d'action commence au premier caractère non vide après le nom de la commande.

6. Autres installations SCP

6.1 Formatage des entrées/sorties du terminal Bibliothèque

Le SIMH fournit des routines pour convertir les caractères d'entrée ASCII au format attendu par VM, et pour convertir les caractères ASCII fournis par VM au format standard C. Ces routines sont les suivantes

> int32 **sim_tt_inpcvt** (int32 c, uint32 mode) - convertit le caractère d'entrée *c* en fonction du *mode*.
> et renvoie le résultat converti (-1 si le caractère n'est pas valide dans le mode spécifié).
>
> int32 **sim_tt_outcvt** (int32 c, uint32 mode) - convertit le caractère de sortie *c* en fonction du *mode*
> et renvoie le résultat converti (-1 si le caractère n'est pas valide dans le mode spécifié).

Les modes pris en charge sont les suivants :

TTUF_MODE_8B	Mode 8b ; pas de conversion
TTUF_MODE_7B	Mode 7b ; le bit de poids fort est masqué
TTUF_MODE_7P	Mode imprimable 7b ; le bit de poids fort est masqué En outre, à la sortie, si le caractère n'est pas imprimable, -1 est renvoyé
TTUF_MODE_UC	Mode majuscules 7b ; le bit de poids fort est masqué En outre, les minuscules sont converties en . Si le caractère n'est pas imprimable, -1 est renvoyé.

En entrée, TTUF_MODE_UC possède un modificateur supplémentaire, TTUF_MODE_KSR, qui force le bit de poids fort à être activé plutôt qu'effacé.

L'ensemble des caractères de contrôle imprimables est contenu dans la variable globale de vecteur de bits **sim_tt_pchar**. Chaque bit représente le caractère correspondant au numéro du bit (par exemple, le bit 0 représente NUL, le bit 1 représente SOH, etc.) Si un bit est activé, le caractère de contrôle correspondant est considéré comme imprimable. Il contient initialement les caractères suivants : BEL, BS, HT, LF et CR. L'ensemble peut être manipulé à l'aide des routines suivantes :

t_stat **sim_set_pchar** (int32 flag, char *cptr) - met **sim_tt_pchar** à la valeur pointée par *cptr* ; renvoie SCPE_2FARG si *cptr* est nul ou pointe vers une chaîne nulle, ou SCPE_ARG si la valeur ne peut pas être convertie ou ne contient pas au moins CR et LF.

t_stat **sim_show_pchar** (FILE *st, DEVICE *dptr, UNIT *uptr, int32 flag, char *cptr) - affiche la valeur de la variable
sim_tt_pchar dans le stream *st*.

Notez que le caractère DEL est toujours considéré comme non imprimable et sera supprimé dans les modes UC et 7P.

6.2 Bibliothèque d'émulation de multiplexeurs de terminaux

Le SIMH permet l'utilisation de plusieurs terminaux. Tous les terminaux, à l'exception de la console, sont accessibles via Telnet. Le SIMH fournit deux bibliothèques de support pour l'implémentation de terminaux multiples : sim_tmxr.c (et son fichier d'en-tête, sim_tmxr.h), qui fournissent des routines de support indépendantes du système d'exploitation pour les multiplexeurs de terminaux ; et sim_sock.c (et son fichier d'en-tête, sim_sock.h), qui fournissent des routines de socket dépendantes du système d'exploitation. Sim_sock.c est implémenté sous Windows, VMS, UNIX et MacOS.

Deux structures de données de base définissent les terminaux multiples. Les lignes individuelles sont définies par un tableau de **tmln**
structures (typedef **TMLN**) : struct

```
tmln {
    SOCKET      conn ;          /* line conn */
    uint32      ipad ;          /* Adresse IP */
    uint32      cnms ;          /* temps de connexion
                                   ms */
    int32       tsta ;          /* État Telnet */
    int32       rcve ;          /* activation du rcv */
    int32       xmte ;          /* xmt enable */
    int32       dstb ;          /* désactivation de TInt
                                   bin */
    int32       rxbpr ;         /* rcv buf remove */
    int32       rxbpi ;         /* rcv buf insert */
    int32       rxcnt ;         /* comptage rcv */
    int32       txbpr ;         /* xmt buf remove */
    int32       txbpi ;         /* xmt buf insert */
    int32       txcnt ;         /* xmt count */
    DOSSIER     *txlog ;        /* xmt fichier journal */
    char        *txlogname ;    /* xmt nom du fichier
                                   journal */
    char        rxb[TMXR_MAXBUF] ;  /* tampon rcv */
    char        rbr[TMXR_MAXBUF] ;  /* rcv break */
    char        txb[TMXR_MAXBUF] ;  /* tampon xmt */
} ;
```

Les champs sont les suivants :

conn	socket de connexion (0= déconnecté)
tsta	État Telnet
rcve	drapeau de validation de la réception (0= désactivé)
xmte	drapeau de contrôle du flux d'émission (0= transmission désactivée)
dstb	Mode Telnet bin désactivé
rxbpr	pointeur de retrait du tampon de réception
rxbpi	pointeur d'insertion du tampon de réception
rxcnt	compte de réception
txbpr	pointeur de retrait du tampon d'émission
txbpi	pointeur d'insertion du tampon d'émission

txcnt	nombre de transmissions
txlog	pointeur sur le descripteur du fichier journal
txlogname	pointeur sur le nom du fichier journal
rxb	tampon de réception
rbr	drapeaux de rupture du tampon de réception
txb	tampon d'émission

L'ensemble des terminaux supplémentaires est défini par la structure **tmxr** (typedef **TMXR**)

```
: struct tmxr {
    int32       lignes ;        /* # lignes */
    int32       port ;          /* port d'écoute */
    SOCKET      maître ;        /* socket maître */
    TMLN        *ldsc ;         /* pointeur sur les descripteurs
                                   de ligne */
    int32       *lnorder ;      /* ordre de connexion des lignes
                                   */
    DISPOSITIF  *dptr ;         /* dispositif de multiplexage */
};
```

Les champs sont les suivants :

lignes	nombre de lignes (constant)
port	port d'écoute principal (spécifié par la commande ATTACH)
master	socket d'écoute principal (rempli par la commande ATTACH)
ldsc	tableau de descripteurs de lignes
lnorder	tableau de numéros de lignes dans l'ordre de la séquence de connexion, ou NULL si l'ordre de connexion défini par l'utilisateur n'est pas requis
dptr	pointeur sur la structure DEVICE du multiplexeur, ou NULL si le dispositif doit être dérivé de l'UNIT transmis au premier appel attach.

Le nombre d'éléments des tableaux **ldsc** et **lnorder** doit être égal à la valeur du champ **lines.** La valeur NULL est attribuée à **lnorder** si la fonction d'ordre de connexion n'est pas nécessaire. Si le premier élément du tableau **lnorder** est -1, l'ordre de connexion séquentiel ascendant par défaut est utilisé. Attribuez la valeur NULL à **dptr** si le périphérique doit être dérivé de l'unité transmise lors de l'appel à **tmxr_attach**.

La bibliothèque sim_tmxr.c fournit les routines suivantes pour prendre en charge les terminaux basés sur Telnet :

int32 **tmxr_poll_conn** (TMXR *mp) - recherche une nouvelle connexion aux terminaux décrits *par mp*. S'il y a une nouvelle connexion, la routine réinitialise tous les états du descripteur de ligne (y compris l'activation de la réception) et renvoie le numéro de ligne (index du descripteur de ligne) pour la nouvelle connexion. S'il n'y a pas de nouvelle connexion, la routine renvoie -1.

void **tmxr_reset_ln** (TMLN *lp) - réinitialise la ligne décrite par *lp*. La connexion est fermée et tous les états du descripteur de ligne sont réinitialisés.

int32 **tmxr_getc_ln** (TMLN *lp) - renvoie le prochain caractère disponible de la ligne décrite *par lp*. Si un caractère est disponible, la variable de retour est :

```
(1<< TMXR_V_VALID)| caractère
```

Si aucun caractère n'est disponible, la variable de retour est 0.

void **tmxr_poll_rx** (TMXR *mp) - recherche les entrées disponibles sur les terminaux décrits par *mp*.

void **tmxr_rqln** (TMLN *lp) - renvoie le nombre de caractères dans la file d'attente de la ligne décrite par *lp*.

t_stat **tmxr_putc_ln** (TMLN *lp, int32 chr) - sort le caractère *chr* sur la ligne décrite par *lp*. Les erreurs possibles sont SCPE_LOST (connexion perdue) et SCPE_STALL (connexion bloquée).

void **tmxr_poll_tx** (TMXR *mp) - demande si la sortie est complète sur les terminaux décrits par *mp*.

void **tmxr_tqln** (TMLN *lp) - renvoie le nombre de caractères dans la file d'attente d'émission de la ligne décrite par *lp*.

t_stat **tmxr_attach** (TMXR *mp, UNIT *uptr, char *cptr) - attache le port contenu dans la chaîne de caractères *cptr* aux terminaux décrits par *mp* et l'unité *uptr*.

t_stat **tmxr_open_master** (TMXR *mp, char *cptr) - associe le port contenu dans la chaîne de caractères *cptr* aux terminaux décrits par *mp*. Cette routine est un sous-ensemble de **tmxr_attach**.

t_stat **tmxr_detach** (TMXR *mp, UNIT *uptr) - détache toutes les connexions pour les terminaux décrits par *mp* et l'unité *uptr*.

t_stat **tmxr_close_master** (TMXR *mp) - ferme le port maître pour les terminaux décrits par *mp*. Cette routine est un sous-ensemble de **tmxr_detach**.

t_stat **tmxr_ex** (t_value *vptr, t_addr addr, UNIT *uptr, int32 sw) - routine d'examen stub, nécessaire parce que les terminaux supplémentaires sont marqués comme attachés ; renvoie toujours une erreur.

t_stat **tmxr_dep** (t_value val, t_addr addr, UNIT *uptr, int32 sw) - routine de dépôt stub, nécessaire car les terminaux supplémentaires sont marqués comme détachés ; renvoie toujours une erreur.

void **tmxr_linemsg** (TMLN *lp, char *msg) - sort la chaîne de caractères *msg* sur la ligne *lp.*

void **tmxr_fconns** (FILE *st, TMLN *lp, int32 ln) - affiche l'état de la connexion au flux *st* pour la ligne décrite par *lp*. Si *ln* est >= 0, la sortie est précédée du numéro de ligne spécifié.

void **tmxr_fstats** (FILE *st, TMLN *lp, int32 ln) - fournit des statistiques de connexion au flux *st* pour la ligne décrite par *lp*. Si *ln* est >= 0, la sortie est précédée du numéro de ligne spécifié.

t_stat **tmxr_dscln** (UNIT *uptr, int32 val, char *cptr, void *mp) - analyse la chaîne pointée par *cptr* pour y trouver un numéro de ligne décimal. Si le numéro de ligne est valide, déconnecte la ligne spécifiée dans le multiplexeur de terminal décrit par *mp*. La séquence d'appel permet à **tmxr_dscln** d'être utilisée comme routine de traitement MTAB.

t_stat **tmxr_set_lnorder** (UNIT *uptr, int32 val, char *cptr, void *desc) - définit le tableau d'ordre de connexion des lignes associé à la structure TMXR indiquée par *desc*. La chaîne indiquée par *cptr* est analysée pour obtenir une liste d'intervalles délimitée par des points-virgules. Les plages sont de la forme :

```
ligne1-ligne2       séquence ascendante de la ligne1 à la ligne2
line1/length        séquence ascendante de ligne1 à ligne1+ length-1 ALL
                    séquence ascendante de toutes les lignes définies
par l'option
                    multiplexeur
```

Le tableau d'ordre des lignes doit contenir un élément int32 pour chaque ligne. La séquence d'appel permet à **tmxr_set_lnorder** à utiliser comme routine de traitement MTAB.

t_stat **tmxr_show_lnorder** (FILE *st, UNIT *uptr, int32 val, void *desc) - affiche l'ordre de connexion des lignes associé au multiplexeur (TMXR *) *desc* sur le flux *st*. L'ordre est affiché sous la forme d'une ligne liste d'intervalles délimitée par des points-virgules. La séquence d'appel permet d'utiliser **tmxr_show_lnorder** comme routine de traitement MTAB.

t_stat **tmxr_show_summ** (FILE *st, UNIT *uptr, int32 val, void *desc) - affiche le résumé de l'état du multiplexeur (TMXR *) *desc* dans le flux *st*.

t_stat **tmxr_show_cstat** (FILE *st, UNIT *uptr, int32 val, void *desc) - fournit les connexions (val= 1) ou les statistiques (val= 0) du multiplexeur (TMXR *) *desc* au flux *st*. Vérifie également si le multiplexeur n'est pas attaché ou si toutes les lignes sont déconnectées.

t_stat **tmxr_show_lines** (FILE *st, UNIT *uptr, int32 val, void *desc) - indique le nombre de lignes dans le multiplexeur terminal (TMXR *) I vers le flux I.

Les simulateurs de terminal ne devraient pas avoir besoin d'accéder aux routines de socket dépendantes du système d'exploitation.

6.3 Emulation de bandes magnétiques Library

Le SIMH permet l'utilisation de bandes magnétiques émulées. Les bandes magnétiques sont émulées comme des fichiers disques contenant à la fois des enregistrements de données et des marqueurs de métadonnées ; le format est décrit en détail dans le document "SIMH Magtape Representation and Handling" (Représentation et gestion des bandes magnétiques du SIMH). Le SIMH fournit une bibliothèque de support, sim_tape.c (et son fichier d'en-tête, sim_tape.h), qui fait abstraction de la manipulation des bandes magnétiques. Cela permet de prendre en charge plusieurs formats de bande, sans modifier les simulateurs de dispositifs magnétiques.

La bibliothèque magtape ne nécessite pas de structures de données particulières. Cependant, elle définit quelques drapeaux d'unité supplémentaires :

MTUF_WLK	l'unité est verrouillée en écriture

Si les simulateurs magtape doivent définir des drapeaux d'unité privés, ces drapeaux doivent commencer au numéro de bit MTUF_V_UF au lieu de UNIT_V_UF. La bibliothèque magtape conserve la position actuelle de la bande magnétique dans le champ **pos** de la structure **UNIT**.

La bibliothèque sim_tape.c fournit les routines suivantes pour prendre en charge les bandes magnétiques émulées :

t_stat **sim_tape_attach** (UNIT *uptr, char *cptr) - Attache l'unité de bande *uptr* au fichier *cptr*. Les simulateurs de bande devraient appeler cette routine, plutôt que la routine standard attach_unit, pour permettre l'extension future du support de format.

t_stat **sim_tape_detach** (UNIT *uptr) - Détache l'unité de bande *uptr* de son fichier actuel.

t_stat **sim_tape_set_fmt** (UNIT *uptr, int32 val, char *cptr, void *desc) - Définit le format de la bande pour l'unité *uptr* au format spécifié par la chaîne *cptr*.

t_stat **sim_tape_show_fmt** (FILE *st, UNIT *uptr, int32 val, void *desc) - Écrit le format de bande pour l'unité *uptr* dans le fichier spécifié par le descripteur *st*.

t_stat **sim_tape_set_capac** (UNIT *uptr, int32 val, char *cptr, void *desc) - Fixe la capacité de la bande pour l'unité *uptr* à la capacité, en Mo, spécifiée par la chaîne *cptr*.

t_stat **sim_tape_show_capac** (FILE *st, UNIT *uptr, int32 val, void *desc) - Ecrit la capacité de l'unité *uptr* dans le fichier spécifié par le descripteur *st.*

t_stat **sim_tape_rdrecf** (UNIT *uptr, uint8 *buf, t_mtrlnt *tbc, t_mtrlnt max) - Lecture en avant de l'enregistrement suivant sur l'unité *uptr* dans le tampon *buf* de taille *max*. Renvoie la taille réelle de l'enregistrement en *tbc*.

t_stat **sim_tape_rdrecr** (UNIT *uptr, uint8 *buf, t_mtrlnt *tbc, t_mtrlnt max) - Lecture inverse de l'enregistrement suivant sur l'unité *uptr* dans le tampon *buf* de taille *max*. Retourne la taille réelle de l'enregistrement en *tbc*. Notez que l'enregistrement est renvoyé dans l'ordre inverse, c'est-à-dire que l'octet 0 de l'enregistrement est stocké dans buf[0], et ainsi de suite.

t_stat **sim_tape_wrrecf** (UNIT *uptr, uint8 buf, t_mtrlnt tbc) - Écriture tampon *uptr* de taille *tbc* comme enregistrement suivant sur l'unité *uptr.*

t_stat **sim_tape sprecf** (UNIT *uptr, t_mtrlnt *tbc) - L'unité spatiale *uptr* avance d'un enregistrement. La taille de l'enregistrement est renvoyée en *tbc*.

t_stat **sim_tape_sprecr** (UNIT *uptr, t_mtrlnt *tbc) - L'unité spatiale *uptr* inverse un enregistrement. La taille de l'enregistrement est indiquée en tbc.

t_stat **sim_tape_wrtmk** (UNIT *uptr) - Écrit une marque de bande sur l'unité *uptr*.

t_stat **sim_tape_wreom** (UNIT *uptr) - Écrit un marqueur de fin de milieu sur l'unité *uptr* (ce qui efface effectivement le reste de la bande).

t_stat **sim_tape_wrgap** (UNIT *uptr, uint32 gaplen, uint32 bpi) - Écriture d'un espace d'effacement sur l'unité *uptr* de
gaplen dixièmes de pouce de longueur à une densité de bande de *bpi* bits par pouce.

t_stat **sim_tape_rewind** (UNIT *uptr) - Rembobine l'unité *uptr*. Cette opération réussit que l'unité soit attachée ou non à un fichier.

t_stat **sim_tape_reset** (UNIT *uptr) - Réinitialise l'unité *uptr*. Cette routine doit être appelée lorsqu'une unité de bande est réinitialisée.

t_bool **sim_tape_bot** (UNIT *uptr) - Retourne TRUE si l'unité *uptr* est au début de la bande.

t_bool **sim_tape wrp** (UNIT *uptr) - Retourne TRUE si l'unité *uptr* est protégée en écriture.

t_bool **sim_tape_eot** (UNIT *uptr) - Retourne TRUE si l'unité *uptr* a dépassé la capacité spécifiée de l'unité spécifiée (conservée dans uptr->capac).

Sim_tape_attach, **sim_tape_detach, sim_tape_set_fmt, sim_tape_show_fmt, sim_tape_set_capac** et **sim_tape_show_capac** renvoient des codes d'état SCP standard ; les autres routines de la bibliothèque magtape renvoient des codes privés pour le succès et l'échec. Les codes d'état magtape actuellement définis sont les suivants :

MTSE_OK opération réussie
MTSE_UNATT l'unité n'est pas attachée à un fichier
MTSE_FMT l'unité spécifie un format de fichier de bande non pris en charge MTSE_IOERR erreur d'E/S du système d'exploitation de l'hôte pendant l'opération MTSE_INVRL longueur d'enregistrement non valide (dépasse le maximum autorisé) MTSE_RECE l'en-tête de l'enregistrement contient un drapeau d'erreur
MTSE_TMK marque de bande rencontrée
MTSE_BOT début de bande rencontré lors de l'opération de marche arrière MTSE_EOM fin du support rencontrée
MTSE_WRP unité protégée en écriture pendant l'opération d'écriture

Sim_tape_set_fmt, sim_tape_show_fmt, sim_tape_set_capac et **sim_tape_show_capac** doivent référencés par une entrée dans la liste des modificateurs du périphérique de bande, comme suit :

```
MTAB tape_mod[]= {
       { MTAB_XTD|MTAB_VDV, 0, "FORMAT", "FORMAT",
         &sim_tape_set_fmt, &sim_tape_show_fmt, NULL },
       { MTAB_XTD|MTAB_VUN, 0, "CAPACITY", "CAPACITY",
         &sim_tape_set_capac, &sim_tape_show_capac, NULL }, ...
       } ;
```

6.4 Prise en charge des points d'arrêt

SCP fournit des mécanismes sous-jacents pour suivre plusieurs points d'arrêt de différents types. La plupart des machines virtuelles implémentent au moins des points d'arrêt pour l'exécution des instructions (type E) ; mais une machine virtuelle peut également permettre des points d'arrêt pour la lecture (type R), l'écriture (type W), etc. Jusqu'à 26 types de points d'arrêt différents, identifiés par les lettres A à Z, sont pris en charge.

L'interface VM avec le paquet de points d'arrêt se compose de trois variables et d'une sous-routine :

> **sim_brk_types** - initialisé par la VM (généralement dans la routine de réinitialisation du processeur) à un masque de tous les points d'arrêt pris en charge.
>
> **sim_brk_dflt** - initialisé par la VM au masque du type de point d'arrêt par défaut.
>
> **sim_brk_summ** - maintenu par SCP, il fournit un résumé sous forme de masque de bits indiquant si des points d'arrêt d'un type particulier ont été définis.

Si la VM n'implémente qu'un seul type de point d'arrêt, **sim_brk_summ** est différent de zéro si des points d'arrêt sont définis.

Pour vérifier si un point d'arrêt d'un type particulier est défini pour une adresse, la VM appelle

> uint32l **sim_brk_test** (t_addr addr, int32 typ) - test pour voir si un point d'arrêt de *type typ* est défini pour l'emplacement *addr* ; renvoie 0 si non, et un masque de bits de tous les points d'arrêt correspondant à typ si oui

La procédure **sim_brk_test** pouvant être longue, elle est généralement précédée d'un test de **sim_brk_summ** :

```
if (sim_brk_summ && sim_brk_test (PC, SWMASK ('E'))) {
       <pause d'exécution> }
```

Pour s'adapter à des schémas de points d'arrêt plus complexes, SCP met en œuvre un concept d'espaces de points d'arrêt. Chaque espace de points d'arrêt est une collection orthogonale de points d'arrêt qui sont suivis indépendamment. Par exemple, dans une simulation multiprocessus symétrique, des espaces de points d'arrêt peuvent être attribués à chaque unité centrale pour distinguer les points d'arrêt E (exécution) des différents processeurs. SCP supporte jusqu'à 64 espaces de points d'arrêt ; l'espace est spécifié par les bits <31:26> de l'argument *typ de* **sim_brk_test**. Par défaut, il n'y a qu'un seul espace de point d'arrêt (espace 0).